Antti Ehrukainen

Tietokannan konversiotyökalu

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

29.4.2013

Metropolia

Tekijä Otsikko Sivumäärä Aika	Antti Ehrukainen Tietokannan konversiotyökalu 38 sivua + 1 liite 29.4.2013
Tutkinto	insinööri (AMK)
Koulutusohjelma	tietotekniikan koulutusohjelma
Suuntautumisvaihtoehto	
Ohjaajat	kehityspäällikkö Arto Ihantoja lehtori Olli Hämäläinen

Insinöörityön ensisijainen tavoite oli tuottaa työkalu, jolla konvertoida olemassa olevan järjestelmän tietokannasta tietosisältö uuden järjestelmän tietokantaan. Toissijainen tavoite oli tehdä työkalu siten, että toteutetulla rungolla olisi mahdollisimman helppo tehdä konversio jonkin toisenlaisen järjestelmän uusimisen tai päivityksen yhteydessä. Lisäksi työhön kuului konversioprosessin strategian suunnittelu.

Työ toteutettiin C#-ohjelmointikielellä .NET-Frameworkin valmiita komponentteja hyväksikäyttäen. Työssä pyrittiin noudattamaan SOLID-periaatteita niin luokkien kuin luotujen kirjastojenkin tasolla. Ohjelmakoodin laatua uudelleenkäytettävyyden kannalta tutkittiin ohjelmistometriikan avulla.

Toteutettu ohjelmisto koostuu karkeasti kahdesta osasta: muuttumattomasta rungosta, jolla voidaan asetuksia muuttamalla käynnistää minkälainen tahansa työssä luotuun malliin perustuvan prosessin toteuttava kirjasto, sekä muuttuvasta prosessin toteutuksesta, joita toteutettiin tässä työssä yksi kappale. Toteutus mukailee suunniteltua konversion strategiaa, joka toimi tässä tapauksessa loppuun asti muuttumattomana.

Ohjelmisto suoriutuu tietokannan konversiosta annetussa ajassa ja tuottaa lisäksi toivottuja tarkistuslistoja, jotta järjestelmän käyttäjät voivat tarkistaa konversion tulokset käyttämättä kohdejärjestelmää. Lisäksi saadut metriikan tulokset tukevat käytännön kokemuksia siitä, että toteutettu runko on uudelleenkäyttökelpoinen.

Avainsanat	tietokannan konversio, ETL, ohjelmistometriikka

Author Title Number of Pages Date	Antti Ehrukainen Tool for database conversion 38 pages + 1 appendix 29 April 2013
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	
Instructors	Arto Ihantoja, R&D Manager Olli Hämäläinen, Senior Lecturer

The main goal of the project described in this thesis was to build a tool with which the data from the current system could be transferred to the database of a newer system. The secondary goal was to build the tool in a way that enables different kinds of conversion processes with minimal effort. In addition, a strategy for the conversion was to be designed.

The software was built in the C# programming language using also pre-existing components of the .NET Framework. During the development of the program, there was an effort to follow SOLID principles at all the levels of the software. Software metrics were used to assure code quality in terms of re-usability.

The program that was made consists roughly of two parts. Firstly, there is a stable frame, which loads concrete assemblies dynamically and executes them according to settings given. Secondly, concrete and unstable implementation of the conversion process was made exclusively for this occasion. The implementation adheres to the designed conversion strategy, which proved successful and working until the end.

The software can handle the conversion in the given timeframe. It also produces checklists for users enabling them to analyze the results without using the new system. The results based on the metrics used support the observations made during development of the software that the produced frame is applicable for reuse.

Keywords	Database migration, ETL, software metrics

Sisällys

Lyhenteet

BI	*Business Intelligence*, tiedon keräämisen, analysoinnin ja hyödyntämisen kautta tapahtuvaa liiketoiminnan hallintaa ja kehittämistä.
C#	Microsoftin kehittämä ohjelmointikieli, joka toimii .NET Framework sovellusympäristössä.
DIP	*Dependency Inversion Principle*. Olio-ohjelmoinnin suunnittelun periaate.
ETL	*Extract, Transform, Load*. BI-prosessi, jonka tarkoituksena on tuottaa raportoinnin tarvitsemaa tietoa erilaisista lähteistä.
ISP	*Interface Segregation Principle*. Olio-ohjelmoinnin suunnittelun periaate.
LINQ	*Language Inetgrated Query,* .NET lisäkirjasto, joka mahdollistaa lambda-kyselyiden suorittamisen olioita ja kokoelmia vasten.
LSP	*Liskov-Substitute Principle*. Olio-ohjelmoinnin suunnittelun periaate.
.NET	*.NET-Framework*. Microsoftin kehittämä sovelluskehys
OCP	*Open-Closed Principle*. Olio-ohjelmoinnin suunnittelun periaate.
SOLID	Viiden olio-ohjelmoinnin suunnitteluun liittyvän perusperiaatteen akronyymi.
SQL	Relaatiotietokantojen yleisesti käyttämä hakukieli.
SQL Server	Microsoftin kehittämä tietokantapalvelinympäristö.
SRP	*Single Responsibility Principle*. Olio-ohjelmoinnin suunnittelun periaate.
SSIS	*SQL Server Integration Services*. SQL Serverin päälle rakennettu graafisen käyttöliittymän datan siirtoon tarkoitettu työkalu.
T-SQL	Microsoftin SQL Server -tietokantapalvelimen käyttämä versio SQL-hakukielestä.

1 Johdanto

Insinöörityön tarkoituksena on tuottaa työkalu, jolla voidaan suorittaa tietokannan konversio. Konversiossa siirretään tietosisältö olemassa olevasta järjestelmästä uuteen järjestelmään. Työlle asetettuja vaatimuksia ovat aikataulussa valmistuminen sekä mahdollisimman hyvä ylläpidettävyys ja muokattavuus tulevaisuuden tarpeita varten. Tärkeysjärjestys vaatimuksille on sama, eli muokkautuvuuden suunnittelu tulee toteuttaa annetun ajan puitteissa. Koska konkreettinen työ tehdään salassapitovelvoitteiden alaisuudessa, tässä raportissa ei käsitellä työnantajan, asiakkaan tai tietosisällön tarkkoja nimiä.

Olemassa olevan erilaisen toteutuksen tietosisältö ei ole rakenteellisesti sopiva toimitettavan uuden järjestelmän tietorakenteen kanssa. Järjestelmän uudistamisen yhteydessä on sovittu vanhan tietosisällön konversiosta, joka on tämän opinnäytetyön konkreettinen tavoite. Toimitettavalle järjestelmälle on myös alustavia suunnitelmia laajemmasta päivityksestä, jonka vuoksi yksi lisätavoite on tuottaa työkalu, joka on mahdollisimman helppo muokata versionnostoa varten sopivaksi.

Teoriaosuudessa pyritään ensin selvittämään, mistä konversiossa on perimmiltään kysymys, ja löytämään ja tutkimaan jo olemassa olevia ratkaisuja. Lisäksi tutkitaan teorioita ja tapoja, joita käyttämällä toinen työlle asetettu tavoite pystyttäisiin toteuttamaan, sekä sitä miten tavoitteiden toteutumisen pystyisi todentamaan.

2 Lähtökohdat

Konversio

Konversio on yleinen ongelma ja tavanomainen tehtävä järjestelmien versionnostojen, integraatioiden sekä järjestelmävaihdosten yhteydessä. Konversion tavoite on siirtää olemassa olevan järjestelmän sisältämä tieto toisenlaiseen tietorakenteeseen, kuitenkaan menettämättä olennaista tietosisältöä. Konversio on siis (kertaluonteinen) prosessi, jonka tarkoitus on mahdollistaa mahdollisimman saumaton ja jatkuva toiminta järjestelmäpäivityksen yhteydessä. [1; 2]

Extract, Transform, Load (*ETL*) [3] on *Business Intelligencessä* (*BI*) tiedon käsittelemiseen käytetty perustyökalu eikä se prosessina juurikaan eroa konversiosta. ETL eroaa kuitenkin tavoitteiltaan ja käyttötavaltaan merkittävästi konversiosta, joten termejä käsitellään tässä työssä erillisinä. ETL:n tarkoitus on kerätä tietoa (useista) tietolähteistä, muokata sitä sopivaksi ja tallettaa se BI-raportoinnin käytettäväksi.

Tietokantojen sisältöjen konversiosta saatavilla oleva tutkimustieto ja teoria on vähäistä työn konkreettisuuden vuoksi, mutta koska ETL ja konversio ovat prosesseina lähes identtisiä, viitteinä käytetään ETL:iin kohdistuvaa tutkimustietoa ja kokemuksia parhaista käytännöistä, joita on saatavilla laajemmissa määrin kuin tietokannan konversiolle. Parhaita ETL-käytäntöjä joudutaan käyttämään harkinnanvaraisesti, sillä työn tarkoitus ei ole tuottaa ETL-prosessia tavanomaisilla ETL-työkaluja, vaan konversiotyökalua räätälijärjestelmänä eivätkä monet ETL:n parhaat käytännöt sovi tämän työn puitteisiin. Tiedon puhdistus ja keräys on myös rajattu toteutettavan työn ulkopuolelle, joten toteutettavan konversion tiedon sisäänluvun osuus on suppeampi kuin mitä se voisi olla.

Ympäristö ja rajaukset

Työn määrää on rajattu osittain jo asiakkaan kanssa tehdyssä sopimuksessa. Asiakas huolehtii tiedon keräyksestä sovittuun muotoon, samoin kuin tiedon suodatuksesta ja relevanttiudesta. Näin saatiin rajattua konversiotyökalun vastuuta ja toteutuksen haasteellisuutta sekä alennettua onnistumiseen kohdistuneita riskejä.

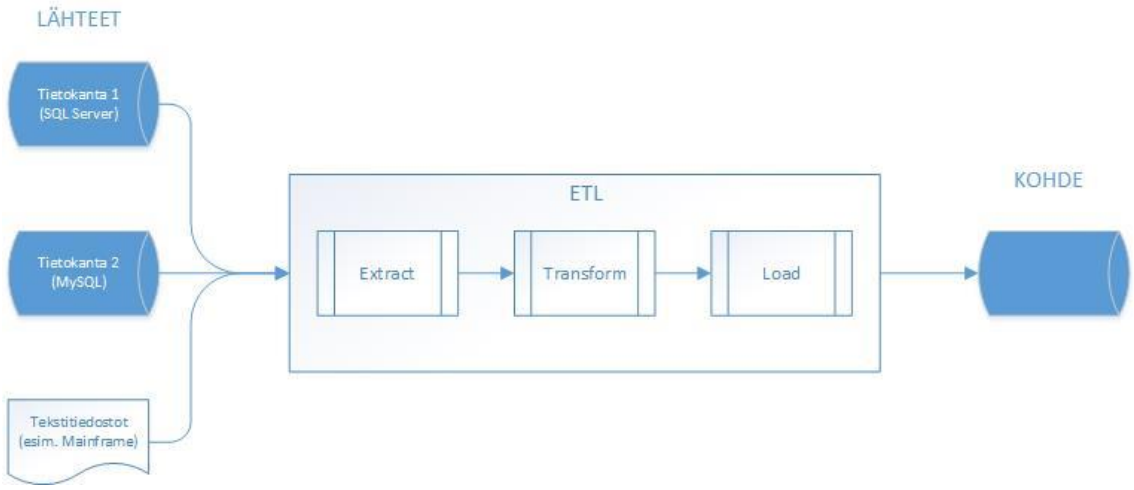
Työ suoritetaan käyttämällä Microsoftin valmistamia sovelluskehitystyökaluja. Samalla käytetään tilaisuutta harjoitella uusimpien työkalujen käyttöä, joten työn toteutuksessa tullaan käyttämään Visual Studio 2012 -sovelluskehitysympäristöä ja .NET-Framework-sovelluskehikon uusinta versiota 4.5. Tiedon kohdejärjestelmä on Microsoftin kehittämä relaatiotietokantapalvelin SQL Server 2008 R2. Tietokannan hallintatyökalu on Microsoftin SQL Server Management Studio 2008. Työssä on lähdetty siitä, että ohjelma käyttää prosessointiin vain yhtä prosessoria.

SQL Serverin ETL-työkaluna käytetään usein *SQL Server Integration Services*-pakettia tai ETL:a varten tehtyä valmisohjelmaa [4; 5]. Tästä huolimatta tämä työ tullaan ohjelmoimaan C#-ohjelmointikielellä. Vaikka päätös valmisohjelman käyttämisestä ei kuulunut työn sisältöön, perustelut päätökselle ovat helppoja ymmärtää. Työn tilanneessa organisaatiossa on sekä paikallisesti kuin muutoinkin enemmän ohjelmisto-osaajia kuin SSIS-osaajia. Uuden teknologian opettelu yhtä työtä varten ei varmasti toisi toteutettavan työn kannalta etuja vaan pikemminkin lisäisi riskejä aikataulun ja onnistumisen suhteen. [5; 6.]

3 ETL-prosessi

ETL ja konversio

ETL-prosessi suoritetaan mahdollisesti useita kertoja päivässä, päivittäin, viikoittain, kuukausittain tai niin usein kuin päivitettyä tietoa tarvitaan raportointia varten [3; 7]. Konversio suoritetaan useimmiten ainoastaan kerran tietylle tietomäärälle valitusta tietolähteestä ja tallennetaan tarvittavien muutosten ja generointien jälkeen kohdejärjestelmään, jotta se olisi kohdejärjestelmän käytettävissä. Konversiossa on olennaista säilyttää mahdolliset tiedon väliset suhteet ja viite-eheydet, jotka eivät aina ole välttämättömyyksiä BI-prosessien kannalta. Konversiossa, toisin kuin ETL:ssa, on myös harvoin tarpeen merkitä tietosisältöön lähdejärjestelmään liittyvää tietoa. Esimerkiksi maantieteellisesti laajalle levinneessä organisaatiossa raportointitiedolle voi olla olennaista tiedon maantieteellinen sijainti, vaikka järjestelmään tallennettu tieto ei sitä sisältäisi.



Kuva 1. Extract, Transform & Load -prosessi.

Kuva 1 havainnollistaa ETL-prosessia. Seuraavassa esittelen ETL-prosessin vaiheet tarkemmin ja peilaan niitä konversion vastaaviin osuuksiin. ETL:n parhaita käytäntöjä käsittelevissä julkaisuissa prosessiin on kuvattu 10–38 eri kohtaa, mutta käyn läpi asiat pääpiirteittäin, sillä osa asioista, eritoten lähdetiedon puhdistus ja analysointi, on rajattu jo valmiiksi pois tämän työn piiristä.

Prosessin ensimmäisessä eli *extract*-vaiheessa haluttu tieto kerätään yhdestä tai useasta lähdejärjestelmästä. Lähdetietoa voidaan hakea niin erilaisista tekstitiedostoista (kiinteän mittaiset rivit, CSV [kentät eroteltu erotin-merkillä] tai XML) kuin erityyppisistä tietokannoista (SQL Server, MySQL) tai web-palvelurajapinnoista. Lähdejärjestelmien mahdollisesta kirjosta johtuen suositus on, että jokaista lähdetyyppiä kohden luodaan yksi osaprosessi ylläpidettävyyden helpottamiseksi. Yritys voi esimerkiksi ostaa toisen yrityksen, jolloin henkilöstötiedot ovat suurella todennäköisyydellä erilaisissa tietojärjestelmissä ja vähintäänkin erilaisissa tietorakenteissa. Kasvavien järjestelmien ylläpitoa helpottaa jokaisen järjestelmän osakokonaisuuden pienenä pysyminen. Tällä tavalla suunnitelluissa järjestelmissä muutos kohdistuu vain pieneen osaan järjestelmää ja ylläpidettävyys paranee, sillä muutoksen tekijän ei tarvitse tuntea tai tutkia koko järjestelmää varmistuakseen, mihin kaikkialle muutos järjestelmässä heijastuu. [7; 8.]

Transform-vaiheen tarkoitus on muokata saatu tieto siten, että sen rakenne on sopiva kohdejärjestelmän kanssa. Vaiheeseen voi kuulua tietosisällön generointia, lisäyksiä tai tarkennuksia mutta olennaisinta on saada merkitsevä tietosisältö sovitun rakenteen mukaiseksi. [7.] Kuten extract, myös transform voidaan toteuttaa jokaista lähdejärjestelmää kohden, jolloin saadaan rajoitettua mahdollisesti muuttuvien osakokonaisuuksien kokoa tilanteessa, jossa lähdejärjestelmän tietoa tarvitsee muuttaa vain vähän.

Load-vaihe lataa kaiken muunnetun tietosisällön ja kirjoittaa sen haluttuun tietosäiliöön. Koska jokainen transform-vaihe tuottaa samanmuotoista tietoa, load-vaiheessa riittää normaalissa tapauksessa vain yksi toteutus. Load on tiedon määrästä ja ympäristöistä riippuen usein hitain yksittäinen osaprosessi, koska se on ainut vaihe, jossa välttämättä kirjoitetaan jonkin I/O-väylän kautta. [7.]

Prosessin osia yhdistävät haasteet

Erinäisten selkeiden osaprosessien lisäksi kokonaisuudessa pitää ottaa huomioon koko prosessiin ja sen ympärillä vaikuttavat yhteistekijät. Ohjelmistotekniikassa näistä käytetään yleisesti nimeä *Cross-Cutting Concerns*, joiden ratkaisemisen helpottamiseen etenkin *Aspect Oriented Programming* [9, s.565–590] pyrkii.

Tiedon välitys eri osaprosessien välillä

Tiedon välityksen ongelma riippuu vahvasti ympäristöstä ja sen asettamista rajoitteista. Valittava tiedonsiirron ratkaisu vaihtelee paikallisesti yhdessä ympäristössä ajettavan prosessin muistinvaraisesta tiedonsiirrosta mannertenvälisen prosessin tietoverkkojen välityksellä tehtävään asynkroniseen tiedonkeräämiseen mahdollisine välitallennuspisteineen. Lisäksi mahdollisia tiedonvälityksen tekniikoita voidaan rajata riskienhallinnan kautta, sillä kaikkea tietoa ei välttämättä haluta julkaista avointen web-rajapintojen kautta. Myös projektin ulkopuoliset tahot voivat asettaa rajoja tekniselle ratkaisulle, esimerkiksi EU:n tietosuojadirektiivi asettaa rajoituksia henkilötietojen käsittelyyn EU:ssa [10].

Diagnostiikka ja virheidenkäsittely

Diagnostiikkaa suunniteltaessa päätetään, mitä kaikkea ja millä tarkkuusasteella prosessin etenemisestä ja osaprosessien tuloksista halutaan tarkkailla. Kuten tiedonvälityksen ratkaisu, tämänkin ongelman ratkaisu riippuu hyvin pitkälle niin teknisestä ympäristöstä kuin prosessin kuluttajan tarpeista. Paikallisen päivittäin tapahtuvan prosessoinnin ei välttämättä tarvitse kirjoittaa erikoisempaa lokia, kun taas laajemman kokoluokan prosessissa on täysin erilaiset tarpeet. Konversio eroaa myös tältä osin ETL:sta, sillä tiedon eheyden kannalta on oleellista tietää tarkasti, paljonko kunkin vaiheen läpikäymästä tiedosta on onnistuneesti käsiteltyä ja ilmoittaa kaikista käsittelyvirheistä jatkotoimenpiteitä varten. [4; 8.]

Virheidenkäsittelyn säännöt todennäköisesti heijastelevat prosessin kriittisyyttä ja käyttävän organisaation sille asettamia suoritus- ja eheysvaatimuksia. Joillakin organisaatioilla voi olla tiukatkin säännöt virheistä selviämiselle, kun taas normaalille yritystoiminnalle raporttien syntyminen minuutilleen oikeaan aikaan ei välttämättä ole olennaista. Konversio eroaa tältä osin paljonkin ETL:sta tiedon eheysvaatimusten takia. [6; 7; 11.]

Prosessiosien tulosten analysointi

Tarpeista riippuen voi olla olennaista joko ainoastaan mahdollistaa ajonaikainen analysointi diagnostiikkatyökalujen avulla, kirjoittaa osaprosessien tuloksia lokiin, tuottaa oma raporttinsa välituloksista tai kirjoittaa välituloksia tietokantoihin uudelleenkäynnistystä tai auditointia varten [6; 11]. Mikäli esimerkiksi virheidenhallinta on vapaamuotois-

ta, voi olla kiinnostavaa tietää tiedon tunnistetietojen lukumäärä ennen osaprosessia ja sen jälkeen. Konversiossa tämä tieto on automaattisesti erittäin oleellista, mutta ETL-prosesseissa se ei ole välttämätöntä tai ainakaan eroavaisuudet luvuissa eivät suoraan kerro prosessin virheellisyydestä. Esimerkiksi myytyjen hyödykkeiden keskinäisen vertailun prosenttiluvut eivät välttämättä juuri muutu, mikäli osa tiedosta ei kulje prosessien läpi. Tällöin on kuitenkin syytä asettaa jonkinlaisia rajoja sille, missä vaiheessa koko prosessin tulos hylätään kattamattomasta otoksesta johtuen. [7; 11.]

4 Toteutukselle asetetut tavoitteet

Toteutettavalle ohjelmalle annettiin ainoastaan kaksi tavoitetta:

1. Ohjelmiston tulisi suoriutua tietosisällön konversiosta.
2. Ohjelmistoa tulisi olla mahdollisimman helppo muokata tulevia konversiotarpeita varten.

Ensimmäinen tavoite on yksinään helppo määritellä, toteuttaa ja todentaa toteutuneeksi. Toisen tavoitteen määritteleminen, toteuttaminen sekä todentaminen on taas huomattavan paljon haastavampaa. Mitä tekniikoita tai tapoja ohjelmistokehitys tarjoaa helpon muokattavuuden kannalta? Mitä tapoja on todentaa helppo muokattavuus, kun toteutetaan ainoastaan yksi versio ohjelmistosta? Mitä määreitä ohjelmiston helppo muokattavuus pitää sisällään?

4.1 Ohjelmiston uudelleenkäyttö

Ohjelmiston uudelleenkäyttöä tapahtuu useilla eri tasoilla sekä luonnollisesti itsestään että harkittuna päätöksenä. Uudelleenkäytön eri tasoja ovat muun muassa metodien uudelleenkäyttö, luokkien uudelleenkäyttö, suunnittelumallien hyödyntäminen, valmiiden ohjelmistokomponenttien käyttö, sovelluskehikoiden (kuten .NET Framework) käyttö, ohjelmistolinjastot (tiettyä liiketoimintaa palveleva ohjelmiston runko, joka räätälöinnin jälkeen otetaan käyttöön eri organisaatioissa) ja valmiiksi olemassa olevien ohjelmistojen käyttö. Itsestään tapahtuva uudelleenkäyttö tapahtuu yksilön tasolla, kun käsillä oleviin ongelmiin sovelletaan aiemmin opittuja taitoja ja ratkaisumalleja. Pisimmälle vietynä harkittuna strategisena päätöksenä käytetään ongelmien ratkaisuun valmisohjelmia, joita asetusten kautta ohjataan omaan käyttöön sopivaksi. [9, s.426–448.]

Työn tavoitteet eivät ole laajemman organisaation strategisia tavoitteita, vaan pikemminkin tähtäävät paikallisella tasolla hyödynnettävissä olevaan tulokseen. Tällöin tuotettu uudelleenkäytettävä hyöty rajoittuu kevyeen sovelluskehikkoon ja kaikkiin sitä pienempiin uudelleenkäytön tasoihin. Näin saadut hyödyt jäävät mahdollisesti pieniksi, mutta vaikutusalue on silti yksilöä laajempi.

4.2 Tavoitteisiin pääseminen

Robert C. Martin kuvailee kirjassaan Agile Principles, Patterns and Practices in C# [12] hyväksi havaittuja olio-ohjelmoinnin suunnittelun perusperiaatteita, joiden avulla pystytään saavuttamaan laadultaan parempaa koodia, jolloin myös muutokset tulevaisuudessa ovat helpompia toteuttaa. Ongelma-alueet, jotka hänen mielestään aiheuttavat eniten ongelmia ohjelmistoissa ja joita hänen esittelemänsä periaatteet pyrkivät parantamaan, ovat seuraavat:

- Jäykkyys: Pienetkin muutokset koodiin saattavat aiheuttaa suurta työtä ja muutoksia paikkoihin, joihin muutosten ei alun perin uskottu tai kuviteltu vaikuttavan mitenkään.
- Herkkyys: Muutokset ja korjaukset rikkovat vanhaa toiminnallisuutta ennalta arvaamattomissa paikoissa ohjelmistoa.
- Liikkumattomuus: Ohjelmiston osia on käytännössä mahdotonta käyttää uudelleen, sillä ne sisältävät niin paljon (turhia) riippuvuuksia, että uudelleenkäyttö toisessa ohjelmistossa on mahdotonta.
- Sitkeys: Ohjelmoijien on helpompaa saavuttaa lyhyen tähtäimen tavoitteet tekemällä asioita alkuperäisen suunnitelman tai arkkitehtuurin vastaisesti.
- Turha monimutkaisuus: Ohjelmisto sisältää luokkia, kutsuja ja rutiineja, joita ei lopulta käytetä missään vaiheessa ja jotka näin tekevät kokonaiskuvan hahmottamisesta vaikeampaa.
- Turha toisto: Koodissa käytetään samantapaista rutiinia asioiden tekemisessä useassa eri paikassa. Tällaisen rutiinin muuttaminen jokaiseen kohtaan aiheuttaa paljon ylimääräistä työtä, etenkin jos rutiinista on useita variaatioita.
- Epäselvyys: Ohjelmakoodi on kirjoitettu epäselvästi, jolloin muutosten toteutus vaikeutuu vaikka kyseessä olisi alkuperäinen ohjelmoija.

Esiteltyjä ohjelmointitason periaatteita on viisi. Lisäksi kirjassa kuvaillaan kirjasto-tason suunnittelumenetelmiä, jotka pitkälti pohjautuvat samoihin periaatteisiin mutta ne esitellään korkeammalla abstraktion tasolla. [12, s.101–107.]

Yhden vastuun periaate

Yhden vastuun periaatteen, eli *Single Responsibility Principlen* (*SRP*) esitteli alun perin Tom DeMarco vuonna 1979 käyttäen termiä *Cohesion* (yhteenkuuluvuus). Periaate on, että jokaisen luokan tulisi olla vastuussa ainoastaan yhdestä asiasta. Tämä johtaa sii-

hen, että jokaisella luokalla on ainoastaan yksi syy muuttua. Näin muutos vaatimuksissa ei johda muutoksiin useissa eri paikoissa järjestelmän sisällä, ainoastaan muutoksen kohde muuttuu. [12, s.115–120.]

Avoin / suljettu-periaate

Avoin / suljettu-periaate (*Open/Closed-Principle, OCP*) Bertrand Meyerin 1988 esittelemän termin mukaisesti ”ohjelmiston tulisi olla suljettu muutokselle, mutta avoinna laajentamiselle”. Tällöin luokkien väliset riippuvuussuhteet ovat ennemmin koostumuksia kuin perimisiä. Tämä vähentää luokkien ja sitä kautta moduulien riippuvuutta toteutuksesta ja antaa mahdollisuuden lisätä tai laajentaa toiminnallisuutta. [12, s.121–133.]

Liskovin korvattavuuden periaate

Barbara Liskov kirjoitti luokkien korvaavuudesta (*Liskov Substitution Principle, LSP*) vuonna 1988. Missä tahansa kohtaa ohjelmistoa, jossa käytetään perittävää luokkaa, tulisi olla mahdollista vaihtaa peritty luokka perittävän luokan tilalle ilman että koodia tarvitsisi muuttaa. Tämä malli auttaa suunnitteluvaiheessa varmistamaan onko luokkien välillä todellista perimissuhdetta vai onko syytä ohjelmoida luokkien käytös rajapintojen tai kompositioiden kautta. [12, s.135–151.]

Rajapintojen erotuksen periaate

Rajapintojen erotuksen periaatteen (*Interface Segregation Principle, ISP*) mukaan rajapintojen vastuiden tulisi olla aivan yhtä tarkasti määriteltyjä kuin luokkienkin. Tämä johtaa siihen, että rajapinnat ovat pieniä ja kuvailevat niitä käyttävien luokkien käytöstä tarkasti. Näin rajapintoja toteuttavien luokkien ei tarvitse muuttua turhaan tai toteuttaa itselleen merkityksettömiä metodeja. [12, s.163–175.]

Riippuvuuden kääntämisen periaate

Riippuvuuden kääntämisen periaatteen (*Dependency Inversion Principle, DIP*) mukaisessa ohjelmakoodissa korkean tason toiminnallisuus ei saisi olla riippuvainen alemman tason toteutuksesta, vaan molempien tulisi olla riippuvaisia abstraktioista sekä yksityiskohtien tulisi olla riippuvainen abstraktioista eikä toisinpäin. Tämä johtaa siihen

että tuotettava koodi kuvailee prosessia paremmin. Yhdessä SRP:n kanssa suunniteltuna ylemmän tason prosessi kutsuu tarkoin rajattuja prosessin osia eri paikoista, jotka eivät liity toisiinsa ja joita voidaan luoda erilaisten tarpeiden mukaan sekä tehdä muutoksia ilman että ylemmän tason prosessia tarvitsee muuttaa. [12, s.153–162.]

Näistä periaatteista käytetään yhdessä esitettynä akronyymiä SOLID. Tämä akronyymi taas muodostuu itse periaatteiden akronyymeista: SRP, OCP, LSP, ISP ja DIP.

4.3 Refaktorointi

Refaktorointi on ohjelmoinnissa käytetty tekniikka, joka pyrkii parantamaan tuotetun koodin ja ohjelmiston rakenteen laatua erityisesti ylläpidettävyyden parantamiseksi. Refaktoroidessa ei muuteta ohjelmiston toiminnallisuutta, vaan muokataan joukolla pieniä muutoksia koodin rakennetta siten että koodia on tulevaisuudessa helppo muuttaa. Martin Fowler nimittää kirjassaan Refactoring [13] ohjelmakoodin kohtia, joita olisi syytä refaktoroida, ’pahoiksi hajuiksi’. Näiden hajujen syyt pyritään poistamaan jo ohjelmointivaiheessa, jolloin korjaaminen on kustannustehokkaampaa kuin jo tuotannossa olevan koodin korjaaminen [14].

Refaktoroinnin perustana toimivaa helppolukuisuutta pyritään lisäämään nimeämällä luokat, metodit ja muuttujat yksiselitteisesti. Kun ohjelmakoodia on helppoa lukea, sen toiminta on helppo omaksua ja tällöin muutokset ja korjaukset on helpompi toteuttaa. Kun koodissa käytetyt nimet kertovat suoraan, mitä nimetyt asiat ovat tai tekevät, on myös helpompi huomata logiikkavirheitä toteutusvaiheessa. Vaikka refaktorointi liittyykin vahvasti ketteriin sovelluskehityksen menetelmiin etenkin oliopohjaisilla kielillä, on refaktorointi hyödyllistä myös muilla ohjelmointikielillä ja muita sovelluskehityksen menetelmiä käytettäessä. [9, s.250–251; 13, s.53–37.]

4.4 Tavoitteisiin pääsyn varmistaminen

Ohjelmiston ominaisuuksia ja laatua on pyritty mittaamaan ja tutkimaan tieteellisesti lähes yhtä kauan kuin ohjelmistoja on tehty. On ehdotettu erilaisia ulkoisia ja sisäisiä mittareita ohjelmiston laadun mittaamiseksi eri kriteerejä käyttäen, esimerkiksi ohjeiden pituus, kirjoitetun ohjelmakoodin rivien määrä ja uuden ominaisuuden kehittämiseen

vaadittu aika. Mittareiden oikeellisuutta ja toimivuutta on yritetty todistaa, mutta edelleenkään ei ole olemassa yksimielisyyttä siitä, että jokin ohjelmakoodin mittari olisi ylitse muiden tai että mittaustulokset olisivat kaikissa tapauksissa täysin pitäviä.

Vaikka ohjelmistometriikka ei antaisikaan täydellistä kuvaa ohjelmiston laadusta, voidaan erilaisia mittareita käyttää suuntaa-antavina tietoina mahdollisista ongelmakohdista. On erittäin todennäköistä, että räikeät ylilyönnit erilaisilla mitta-asteikoilla viittaavat todellisiin ongelmiin. Mikään funktio, jonka syklomaattinen kompleksisuus (*Cyclomatic complexity*, metodin mahdollisten suoritustapojen määrä) ylittää 50, ei todennäköisesti ole helposti ymmärrettävissä tai ylläpidettävissä, mutta se ei tarkoita, että joskus harvoin ei olisi perusteltua kirjoittaa monimutkaista ja pitkää koodia tai että mittaustulokset viittaisivat oikeaan ongelmaan. On kuitenkin syytä ymmärtää ohjelmistometriikkaa ja käyttää mittaustuloksia osoittamaan tuotetun koodin laadullisia ominaisuuksia. Rajaarvojen ylittäminen ei suoraan kerro huonosta laadusta, mikäli ratkaisu on arkkitehtuurillisesti perusteltu. [9 s.668–678; 12 s.435.]

5 Prosessin suunnittelu

5.1 Kohde- ja lähdejärjestelmien tietorakenne

Työn alkaessa oli olemassa jo kohtalaisen selkeä kuva kohdejärjestelmän käyttämästä tietorakenteesta ja toiminnasta, joka oli saatu työskentelemällä järjestelmän parissa. Tästä oli huomattava apu työtä aloittaessa, mutta mitä pidemmälle työ eteni, sitä vähemmissä määrin alussa ollut tietotaito oli avuksi erinäisten kohdejärjestelmän tietorakenteesta johtuvien ongelmien noustessa esiin.

Saapuvan lähdetiedon rakenteesta ei ollut ensin varmuutta, joten koko ohjelmaa ei voitu tehdä valmiiksi ennen testitiedostoja, vaikka lähdejärjestelmän tietuekuvaukset olivatkin jo tiedossa. Vasta kun varmistui, että lähdejärjestelmän tieto tulee käytettäväksi tasapituisena tekstitiedostona, pystyttiin ohjelmaan tietueiden sisään lukeminen toteuttamaan.

5.2 Tietueiden kuvaus

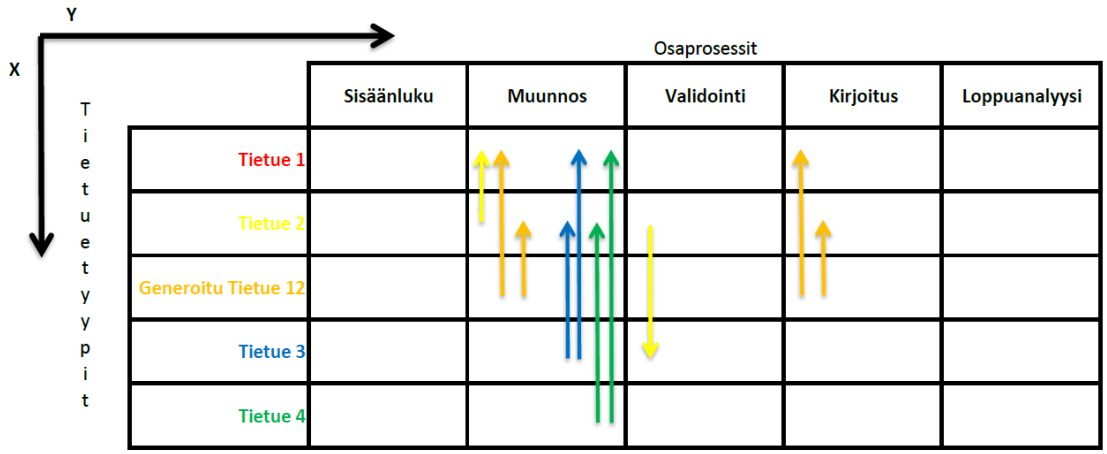
Kaikkein tärkein osuus valmistelevasta työstä oli lähdejärjestelmän sisältämän tiedon kohdistaminen kohdejärjestelmän tietorakenteeseen. Kun lähdejärjestelmän tietuekuvaukset oli saatu, kuvattiin tietueiden tiedot kuvausten perusteella. Jokaisen tiedon nimi, tietotyyppi, pituus, kohdejärjestelmän vastaavan tiedon nimi, tietotyyppi, pituus ja tarpeelliseksi koetut huomautukset kirjattiin taulukkoon. Tämän jälkeen listalle lisättiin kohdejärjestelmän tiedot, joille ei suoraan löytynyt vastinetta lähdetietueista ja huomautuksiin joko tiedon vakioarvon tai arvion mistä tieto olisi saatavilla. Tämän jälkeen vielä jokainen rivi korostettiin väreillä haasteellisuutensa perusteella.

Tämä työvaihe palveli useaa eri tarkoitusta: pienellä vaivalla saatiin kokonaiskuva vaadittavasta työstä, lisäselvitystä vaativista kohdista, toteutusta varten valmis lista lähdetietueiden purkamisesta ja kohdistamisesta sekä dokumentaatio niin sisäisesti kuin asiakkaallekin vahvistettavaksi.

Sovellusta tehdessä ilmeni muutama kohta lisää, joita listalla olisi voinut olla tekemisen helpottamiseksi. Nämä tiedot olivat tiedon järjestysnumero sekä tiedon alkupositio si-

sään luettavassa tiedostossa. Näitä ei enää kirjattu olemassa olevaan listaan, mutta niistä olisi ollut apua toteutusta kirjoittaessa.

5.3 Konversion strategia



Kuva 2. Tietueiden väliset riippuvuudet osaprosesseittain jaettuna.

Kuvassa 2 on esitelty tässä konversiossa muunnettavien tietueiden riippuvuussuhteet osaprosesseittain. Värejä on käytetty korostamaan sitä, mikä on riippuvainen tietue. Nuolen kärki osoittaa tietuetta, johon riippuvuus on. Nuolet ovat aina sen sarakkeen (vaiheen) kohdalla, jossa riippuvuus käytännössä realisoituu. Tietueet on järjestetty riippuvuussuhteiden perusteella järjestykseen, vähiten riippuvainen tietue ylimpänä ja eniten riippuvainen (tässä tapauksessa Tietueet 3 ja 4 ovat yhtä riippuvaisia) alimpana. Tämä helpottaa kokonaisprosessin ymmärtämistä.

Muunnettavien tietueiden riippuvuussuhteet, muunnettavan tiedon määrä sekä muunnettavien tietuetyyppien määrä vaikuttivat konversion strategian suunnitteluun. Vaihtoehtoina olisi ollut tietueiden käsittely yksi kerrallaan koko prosessin läpi (ensin y-akseli loppuun yhden x-akselin askeleen aikana), vaihe kerrallaan kaikki tietueet läpi (kaikki x-akselin osat yhden y-akselin askeleen aikana) tai jokin edellä mainittujen lähestymistapojen yhdistelmä.

Tästä eteenpäin ensimmäistä vaihtoehtoa kutsutaan tietuelähtöiseksi strategiaksi ja toista vaihtoehtoa prosessilähtöiseksi strategiaksi. Lisäksi voisi käyttää edellisten yhdistelmää esimerkiksi siten, että ennen tiedon kirjoittamista kohdejärjestelmään edet-

täisiin tietuelähtöisen strategian mukaan, mutta lopullinen kirjoitus tehtäisiin prosessilähtöisen strategian mukaisesti.

Tietuelähtöisen strategian etuna on mahdollisuus rajata prosessoitavan tiedon määrää helposti, jolloin samalla pystytään hallitsemaan hetkellisesti tarvittavan muistin maksimimäärää. Tällöin ohjelmalle annettaisiin esimerkiksi maksimi tietuemäärä yhtä erää kohden ja ohjelma pilkkoisi sisään luettavaa tietoa automaattisesti sopiviin eriin. Tietuelähtöisen strategian heikkous on riippuvuuksien aiheuttama ylimääräinen työ. Jos tietue riippuu muista tietueista, näihin tietueisiin tulee olla prosessointivaiheessa pääsy. Jos taas aiemmin prosessoitu tietue riippuu myöhemmin prosessoitavasta tietueesta, tulee myöhemmin prosessoitavan tietueen prosessilla olla mahdollisuus muokata jo aiemmin tallennettua tietuetta/tietueita. suosittelisin tämän strategian käyttämistä, mikäli erilaisia tietueita on useampia tai muunnettavaa tietoa on paljon.

Prosessilähtöisen strategian selkeä etu on se, että kun yksi osaprosessi on suoritettu, se on samalla täysin valmis ja sen lopputulema voidaan tallentaa, jotta siihen voi palata myöhemmin. Toinen etu on se, että koska kaikki osaprosessin tarvitsema tieto on kerralla muistissa, tiedon hakeminen ja muokkaaminen on nopeampaa kuin jatkuvat I/O-tapahtumat. Tämän strategian heikkous on virhetilanteiden käsittely. Mikäli tilaa ei osaprosessien välissä tallenneta ja ohjelmassa tapahtuu vakava virhe tietoa tallennettaessa, joudutaan koko prosessi aloittamaan alusta. [6.] Voi myös olla haastavaa käsitellä kaikkien tietueiden kaikkia riippuvuuksia kerralla muistissa, mikäli riippuvuuksia tietueiden välillä on paljon. Tällöin voi olla helpompaa toteutuksen kannalta käsitellä selkeästi yksi tietue kerralla haluttuun pisteeseen asti.

5.4 Valittu strategia

Työn lähtökohdaksi valittiin prosessilähtöinen strategia, koska erilaisia tietueita lähdejärjestelmästä oli vain neljä, niiden väliset riippuvuudet olivat vähäisiä ja selkeitä sekä siksi, että tietueiden kappalemäärä oli vähäinen, alle puoli miljoonaa kappaletta. Lisäksi suoritettava konversio liittyy kerralla tehtävään järjestelmäpäivitykseen, jolloin prosessin tulisi olla mahdollisimman nopea, jotta mahdollisimman pian tiedetään tarve tehdä palautus virhetilanteen sattuessa. Tiedon eheys on olennaista ja vanhaan järjestelmään ei syötetä mitään tietoa sen jälkeen kun konversion lähdetietoa aletaan kerätä. Lähdejärjestelmän käyttöä jatketaan, jos konversiossa tapahtuu virhe tuotantoon siirron

aikana, esiintyneet ongelmat ratkotaan ja yritetään tuotantoon siirtoa myöhemmin uudelleen.

6 Ohjelmiston valmistaminen

6.1 Ohjelmiston suunnittelu

Ohjelmiston korkean tason suunnittelu aloitettiin samoihin aikoihin työn tekemisen valmistelujen ja lähdemateriaalien tutkimisen kanssa. Koska kyseessä on ohjelmiston ensimmäinen versio ja kehittäjiä on ainoastaan yksi, mihinkään malliin ei sitouduttu ennen konkreettisen tekemisen aloittamista. Ohjelmiston arkkitehtuuria muutettiin vielä kertaalleen sen jälkeen kun kaksi eri prosessin vaihetta oli valmiina.

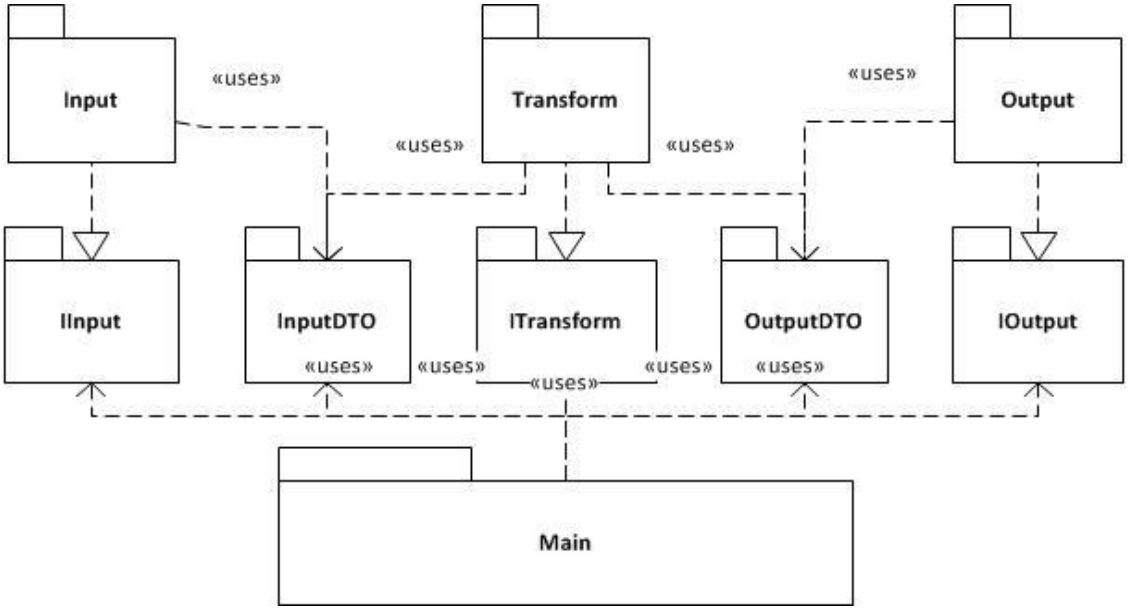
Tietämys ohjelman vaaditusta toiminnallisuudesta lisääntyy tehdessä, eikä toiminnallisuuksia suunniteltu siksi tarkasti etukäteen. Ensin toteutettiin toimiva versio ja vasta sitten tarkasteltiin ja mahdollisesti korjattiin tehdyn toteutuksen laatua. [9 s.43–44; 13 s.67–68; 14.]

Ohjelmiston suunnittelua aloitettaessa vaatimus helposta uudelleenkäytöstä vaivasi todella paljon. Ohjelmiston tekeminen siten, että se täyttää annetut vaatimukset, ei välttämättä sinällään ole haastavaa, etenkin jos annetut vaatimukset ovat yhteä selkeät kuin tässä tapauksessa. Kyseessä on kuitenkin ”ainoastaan” tiedon lukeminen lähteestä ja tietokantaan tallentaminen muokkauksen jälkeen.

Lähdemateriaaleissa, joissa käsiteltiin ohjelmiston uudelleenkäyttöä, yksi ensimmäisistä vastaan tulleista tekniikoista oli komponenttipohjainen ohjelmistokehitys. Jokainen ohjelmiston osa olisi yksi komponentti, jolloin yksittäisiä osia voisi käyttää uudelleen. [9 s.452–476.] Sama teema toistui muissakin lähteissä, joskin mahdollisesti eri tavalla muotoiltuna [12 s.426–430]. Näitä pohjatietoja käyttäen ohjelmiston rakennetta alettiin hahmotella.

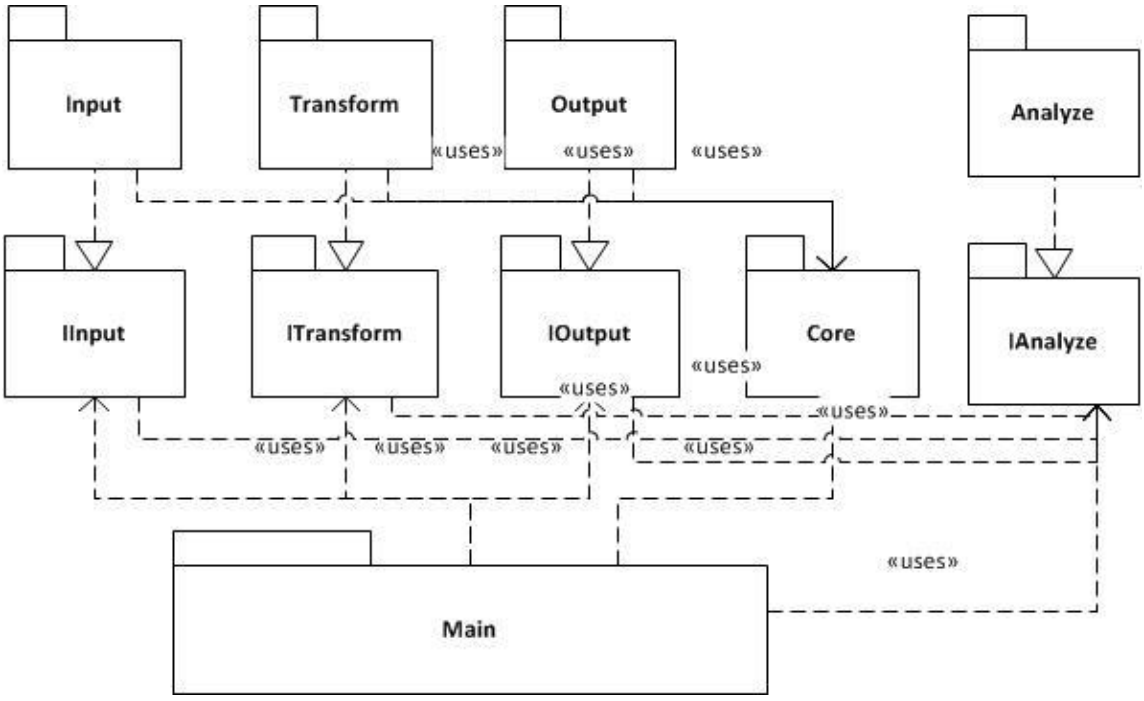
Tutkittaessa komponenttipohjaisen mallin mahdollista toteutusta tarkemmin kävi pian selväksi, että pelkästään tekemällä jokaisesta konversioon liittyvästä vaiheesta oman toteutuksensa toteutettu ohjelmisto ei siltikään olisi välttämättä käytettävissä uudelleen. Seuraavassa konversiossa muunnettava tieto saattaisi tulla jostain muuntyyppisestä lähteestä, jolloin sekä sisäänluku, muunnos että konversion ympärille rakennettava ohjauskin pitäisi kirjoittaa uudelleen, sillä esimerkiksi välttämättä ei olisi kovinkaan kannattavaa lukea tietokantaa tekstimuotoon ja muuttaa se jälleen takaisin tietokantamuo-

toon. DIP:n määritelmä ”Korkean tason moduulien ei pitäisi riippua matalan tason moduuleista, molempien pitäisi olla riippuvaisia abstraktioista.” tarjosi tähän ratkaisun, jonka jälkeen ensimmäinen raakaversio siitä, minkälainen ohjelmiston runko voisi olla, saatiin hahmoteltua. Sama lause konversion kontekstissa esitettynä: ”Konversion ohjauslogiikan ei pitäisi olla tietoinen vaiheiden toteutuksesta, vaan molempien tulisi käyttää määriteltyä rajapintaa”.



Kuva 3. Ensimmäinen versio ohjelman rungosta.

Kuva 3 esittää ensimmäistä hahmotelmaa ohjelmiston rungoksi. Vaikka ratkaisu ei ihanteellinen ollutkaan (sekä ohjauslogiikka että osaprosessit ovat riippuvaisia konkreettisista luokista, jotka sisältävät osaprosessien tarvitseman alku- ja lopputiedon), oli se silti askel eteenpäin. Tämä malli kuitenkin korostaa uudelleenkäytön kannalta oleellista asiaa: mikään osa-alue ei ole toisistaan suoraan riippuvainen. Näin sisäänluvun toteutuksen muutokset eivät vaikuta tiedon kirjoittamiseen eivätkä välttämättä muunnokseenkaan. Ensimmäisen suunnitelman version (joka sisälsi välttämättömät perusteet) valmistuttua jatkettiin eteenpäin pohtien samalla, miten osa-alueita yhdistäviä ongelmia voisi toteuttaa tällä rungolla.

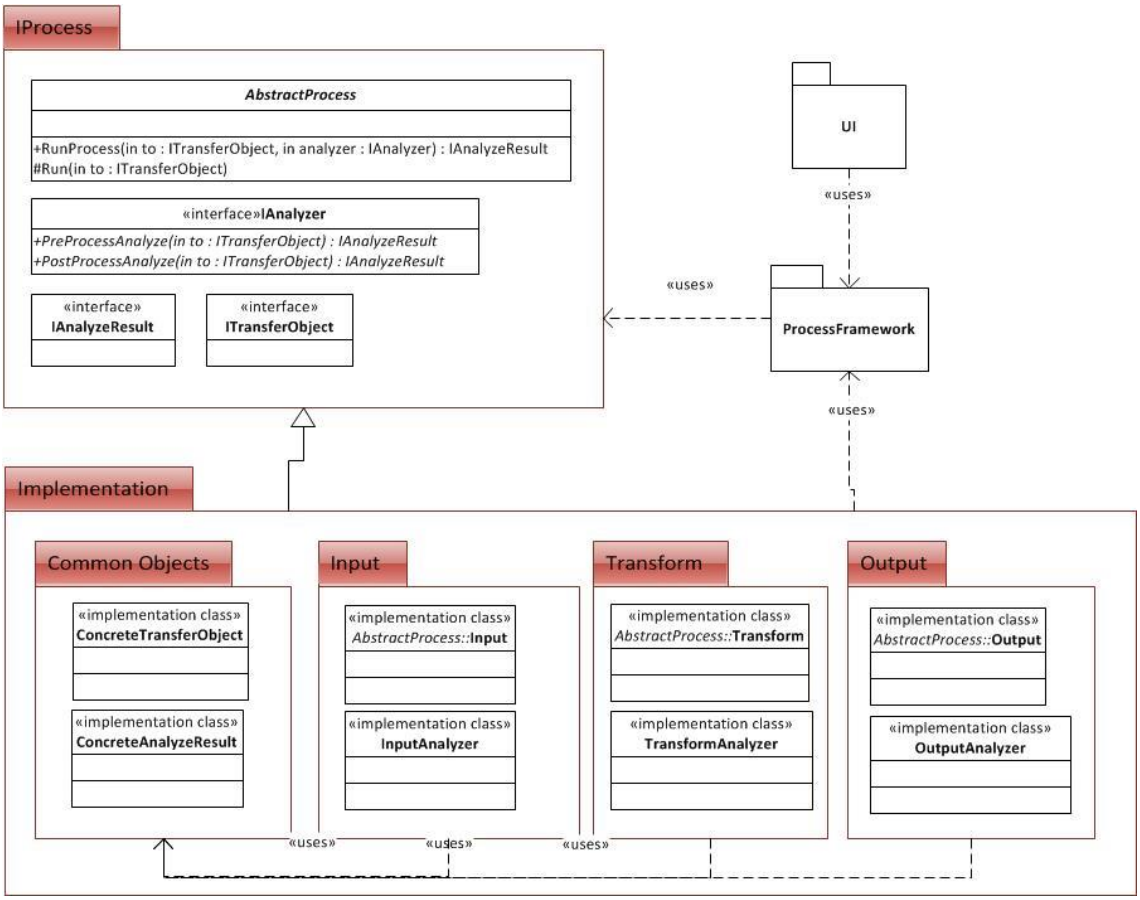


Kuva 4. Toinen versio ohjelman rungosta.

Kuten kuvasta 4 voi huomata, valittu rakenne tukee vaatimusten muutoksia huonosti. Uuden toiminnallisuuden lisääminen vähensi rakenteen selkeyttä, vaikka osaprosessien riippuvuuksia yhdistettiinkin siten, että ne riippuisivat yhdestä moduulista, joka sisältäisi kaiken niiden yhdessä käyttämän tiedon.

Versiohahmotelman jälkeen tietuekuvauksia lähdejärjestelmästä alkoi saapua ja sisäänluku sekä muunnos ehdittiinkin toteuttaa toisen version mallin mukaisesti. Seuraavan suunnitelman version muodostumista auttoi Rich Hickeyn esitelmä ”Hammock Driven Design” ja siinä jaetut kokemukset ongelmien ratkaisusta ja järjestelmien suunnittelusta (joskin skaala on tässä työssä hieman eri luokkaa), joissa hän kehottaa ajattelemaan ongelmaa ennen ratkaisun yrittämistä aktiivisesti hereilläolon aikana ja lepäämään hyvin, jolloin myös alitajunnalla on mahdollista auttaa ongelman ratkaisussa [14]. Suunnitelmia ja prosessimallinnuksia tarkastellessa ja parempaa ratkaisua miettiessä jostain asiayhteydestä tai tekstistä tarttui erityisesti sana prosessi. Tarkempi tarkastelu tältä kannalta varmisti, että toteutuksessa käsitellään prosessia, joka koostuu osaprosesseista, joilla jokaisella on selkeä alku ja loppu. Näin löytyi jokaista erillistä konversioon liittyvää osaprosessia yhdistävä tekijä ja päästiin eroon usean eri rajapinnan yksityiskohtaisesta määrittelystä ja ylläpidosta.

Rungon uudelleenkäytön erityisvaatimuksineen pyöri myös suunnitelmissa mukana. Eräs näistä vaatimuksista on se, että ohjelmaa ei tarvitsisi kääntää uudelleen, vaikka konversion toteutus muuttuisikin. Jälleen kerran DIP ratkaisi ongelman. Yhteen ohjelmakirjastoon sisällytettiin prosessin, prosessituloksen, analysaattorin ja analysointituloksen rajapinnat. Tässä vaiheessa käytiin vielä läpi mitä kaikkea yhteistä eri osaprosesseilla on. Toistuva teema tarkemmassa prosessiketjussa oli mahdollistaa tulosten analysointi osaprosessien välissä. Prosessin rajapinta muutettiin abstraktiksi luokaksi, johon määriteltiin analysaattorin käyttö sekä ennen että jälkeen prosessin suorituksen. [15, s.325–330.]



Kuva 5. Kolmas ja viimeinen ohjelmiston rungon versio.

Kuvassa 5 näkyy, miten luokat on järjestelty moduulien sisällä. Jokaista osaprosessiluokkaa varten tulee olla oma analysaattoriluokkansa, jotta jokaisen osaprosessin toteuttavien luokkien tarvitsee tietää ainoastaan omasta osaprosessistaan. Periytymistä ja implementointia kuvaavia nuolia ei ole piirretty kuvaan standardilla tavalla, jotta ajatus toiminnasta välittyisi paremmin ja kuva olisi selkeämpi. Sovellusta varten luodussa

kehikossa (ProcessFramework) on yleisiä luokkia, eikä niiden tarkempi esittely anna lisäarvoa ratkaisun kuvaukselle tässä vaiheessa. Viimeisessä versiossa on ratkaistavaa ongelmaa kohden selkeästi eroteltu prosessin mukaan muuttuvat osat (prosessin implementaatio) muuttumattomista osista (Käyttöliittymä, sovelluskehikko, prosessin rajapintoja kuvaileva ohjelmistokirjasto) ja ratkaisun tuottaman rungon uudelleenkäyttömahdollisuudet paranivat. Tällä ohjelmiston rungolla pystyisi siis periaatteessa toteuttamaan minkä tahansa prosessin (ei pelkästään konversio tai ETL), jolla on selkeä alku ja loppu, tai jonka osaprosesseilla on selkeät alut ja loput. [9, s.425–434; 12, s.417–418.]

6.2 Ohjelmistometriikan käyttö työssä

Martin esittelee kirjassaan [12, s.455–457] myös ohjelmistomittareita, joilla pystytään mittaamaan periaatteiden noudattamista. Koska konversiotyökalun vaatimuksiin kuului ohjelmiston ylläpidettävyys, alkuperäinen ajatus oli mitata ohjelmasta kehityksen eri vaiheissa kyseisiä arvoja ja käyttää näitä apuna löytämään ohjelmistosta kohtia, joita voisi parannella. Aiheen lähempi tutkiminen johti kuitenkin siihen, että mittauksia esitellyillä mittareilla ei suoriteta kuin enintään työn päätyttyä. Tämä johtuu siitä, että C#-kielelle tai .NET Frameworkille ei ole olemassa ilmaista metriikkatyökalua, jolla pystyisi mittaamaan kyseisiä määreitä. Kehityksen aikana on kuitenkin käytetty Visual Studio 2012:n sisäänrakennettua metriikkatyökalua, jolla voi mitata koodista klassisempia arvoja, kuten rivien määrää (*Lines of Code*), systemaattista kompleksisuutta (*McCabe's Cyclomatic complexity*), ylläpidettävyysindeksiä (*Maintainability Index*) ja olioiden sukupuun syvyyttä (*Depth of Inheritance Tree*) [9, s.668–672].

Lähteessä 12 mainituilla mittareilla tullaan mittaamaan kerran maksullisen työkalun koeajalla (NDepend [19]) ohjelmiston lopullisesta versiosta, mutta tällöin täytyy ottaa huomioon, että näiden mittausten tuloksia ei ole pystytty hyödyntämään kehityksen aikana, ainoastaan toteamaan onko lopullinen tuote yleisesti hyvinä pidettyjen rajojen sisällä.

6.3 Sovelluskehikon toteutus

Seuraavaksi esitellään ratkaisun teknistä toteutusta asiakokonaisuus kerrallaan olennaisina koettujen osien ja ominaisuuksien osalta. Mahdollisia alkuhaasteita sekä toteutukseen johtaneita perusteluita pyritään selittämään ja joissain tapauksissa käydään lopullinen ratkaisu tarkemmin läpi ohjelmakoodin osia hyväksi käyttäen.

Kohdejärjestelmän toiminta on ennestään tuttua, joten jo valmiiksi oli kohtalainen aavistus siitä, minkätyyppisiä yleisiä apuluokkia tullaan tarvitsemaan. Ensimmäiseen versioon otettiinkin aiemmin toteutetuista projekteista kokonaisia luokkia, joista arveltiin olevan hyötyä sellaisenaan.

Lopulta muutamia luokkia poistettiin kokonaan käyttämättöminä. Poistetut luokat liittyivät ohjelmiston instrumentointiin sekä lokin kirjoittamiseen. Lokin kirjoitukseen liittyvän luokan poisto oli kenties hieman hätiköityä, sillä tällä hetkellä I/O-operaatiosta vastaava luokka peri tämän toiminnallisuuden. Toisaalta tässä ohjelmistossa analyysitulosten kirjoittamisen ja yleisen lokin kirjoittamisen raja on häilyvä, eikä enää loppumetreillä ollut syytä alkaa muokkaamaan sovelluskehikkoa.

Sovelluskehikon vastuisiin kuuluvat I/O-operaatiot, asetusten hallinta, tietokantaoperaatiot, tietotyyppien muunnokset sekä prosessinhallinta. Tässä moduulissa olevien luokkien on yritetty pysyä mahdollisimman yleisellä tasolla, jotta luokkia voisi käyttää uudelleen myöhemminkin. Mikäli sovellus olisi huomattavasti laajempi, olisi edellä mainitut vastuut hyviä kohteita kokonaan omiksi kirjastoikseen tai kirjaston sisällä muutamien luokkien kokoisiin nimiavaruuksiin, mutta näin pienessä projektissa omiin luokkiin jako koettiin riittäväksi.

ProcessManager

ProcessManager on luokka sovelluksen kehikosta, joka kaipaa tarkempaa esittelyä, sillä tämän luokan avulla voidaan ottaa käyttöön mikä tahansa prosessi, joka on yhdessä kirjastossa ja jonka jokainen osaprosessi noudattelee tässä sovelluksessa esiteltyjä abstraktioita. Näin luotu ohjelmiston runko ei ole millään tavalla riippuvainen toteutuksesta, vaan samaa runkoa hyväksikäyttäen pystytään kirjoittamaan myös esimerkiksi konversio, jonka lähtökohta olisi tietuelähtöinen strategia.

Ohjelmiston asetusten kautta saadaan tietoon käytettävän implementoivan kirjaston koko nimi. Kirjaston täytyy olla samassa kansiossa kuin käynnistävän ohjelmiston. Tämän jälkeen samojen asetusten kautta saadaan lista osaprosessien nimistä. Jokaisen abstraktiota toteuttavan osaprosessin luokkien tulee noudattaa samanlaista nimeämistä, jossa osaprosessin nimi sisältyy aina luokan nimeen. Kaikille osaprosesseille yhteisten luokkien ei tarvitse noudattaa tätä nimeämistä.

ProcessManager käynnistää annetusta kirjastosta nimeämisskeemaa noudattelevat luokat .NET Frameworkin *reflection* -tekniikkaa käyttäen. Reflectionin avulla voidaan ’käynnistää’ luokkia valmiiksi käytetystä kirjastosta dynaamisesti, eli käynnistävän prosessin ei tarvitse olla millään tavalla tietoinen käynnistettävästä luokasta tai sen toiminnasta. Käynnistetyt luokat säilötään taulukkoon asetuksissa esitetyn prosessijärjestyksen mukaan järjesteltynä. Näin prosessi voidaan ajaa joko kokonaan tai osa kerrallaan. Tämä rajaa ajettavan prosessin selkeisiin vaiheisiin, joiden välissä on mahdollisuus analysoida tuloksia ja tallentaa prosessin tila myöhempää käyttöä varten. Liitteessä 1 on ProcessManager-luokan ohjelmakoodi osaprosessien latauksen osalta.

Tietotyypit ohjelman sisällä

Yksi implementaation kannalta olennaisimpia asioita oli päättää konversion sisällä tapahtuvan tiedonsiirron tietotyypit, sillä vaikka eri osaprosessit ovatkin toisistaan riippumattomia, täytyy jokaisen osaprosessin käsitellä edellisen vaiheen tulosta. Tämä muodostaa eri vaiheiden välille epäsuoran riippuvuuden. Erilaisia vaihtoehtoja toteutuksen kannalta on vähintään:

- Ei erityistä tyypitystä tiedolle, luodaan tarvittavat T-SQL -lausekkeet kirjoituksen aikana ja tallennetaan tiedot tietue kerrallaan.
- Käytetään tietokantaa kuvaavaa tyypitystä, jolloin tieto tallennetaan muistissa muotoon, joka vastaa tietokannan rakennetta. .NET Frameworkista löytyy valmiit luokat tätä varten.
- Käytetään *Object-relational Mapperia* (ORM), joka generoi tietokannan tauluja vastaavat luokat valmiiksi oikein tyypitettyinä.

Näistä kolmesta vaihtoehdosta valittiin keskimmäinen, sillä se antoi riittävän paljon tukea tekemiselle kuitenkaan monimutkaistamatta ohjelmistoa yhtään enempää kuin oli tarvetta. Ensimmäinen vaihtoehto olisi asettanut liikaa rajoitteita virheidenhallinnan,

tietosisällön analysoinnin ja tilan tallentamisen kannalta. ORM:n käyttö taas olisi tuonut tarpeettomia ominaisuuksia ohjelmiston rakenteeseen. Kokemusta kohdejärjestelmän käyttämisestä Entity Frameworkin kanssa oli jo ennestään, eivätkä aiemmatkaan kokemukset tukeneet käytön järkevyyttä, saati sitten tällä kertaa, kun kyse oli ainoastaan tiedon muuntamisesta, eikä käsiteltävän tiedon tarvinnut esittää olioita.

Ratkaisu tuntui erittäin toimivalta. Vaikka aikaa pitikin käyttää etukäteen tietokannan skeeman koodiksi muuntamiseen, tällä tavalla pystyttiin kiertämään haasteet viiteeheyksien kanssa, sillä virheelliset rivit jäivät kiinni jo muunnostilanteessa eivätkä vasta kohdejärjestelmään kirjoitettaessa. Tällä tavoin virheellinen sisään tuleva tieto voidaan ottaa talteen jo ajoissa ajon aikana, joka taas mahdollistaa päätöksenteon konversioajon loppuun viemisestä ilman, että tietokantaan tarvitsee edes ottaa yhteyttä. Tietosisällön säilömiseen käytetyt luokat DataSet, DataTable ja DataRow löytyvät .NET Frameworkin System.Data.Sql-kirjastosta

Tiedonsiirto eri osaprosessien välillä

Osaprosessien välinen tiedonsiirto toteutettiin omana olionaan. Perusteena oli sekä se, että näin abstraktin kirjaston ei tarvitse tietää siitä, mitä implementaation osaprosessit viestittävät toisilleen, kuin että tällä tavalla on mahdollista kirjoittaa osaprosessin tulos tiedostoksi levyjärjestelmään mahdollista myöhempää käyttöä varten. Välituloksen tallentaminen esiintyi suunnitelmissa jo melko aikaisin, mutta se toteutettiin viimeisten asioiden joukossa. Toteutus ei varmastikaan olisi ollut yhtä kivutonta kuin mitä se oli, ellei asiaa olisi otettu huomioon jo aikaisessa vaiheessa.

Tässä työssä esiintyvässä implementaatiossa tiedonsiirto-olio ei tiedä omasta tilastaan mitään, mutta se olisi täysin mahdollista toteuttaa. Jokaisen osaprosessin valmistumisen jälkeen tiedonsiirto-olio tallennetaan kansioon käsitellyn osaprosessin nimellä. Tällainen lähestymistapa on ollut riittävä tätä työtä varten.

Virheidenkäsittely

Perinteinen virheidenkäsittely toteutuksessa on lähes olematonta, sillä prosessin kriittisyydestä johtuen virheiden on parempi tulla käyttäjälle ilmi mahdollisimman pian. Kaikki

virhetilanteet, joihin ei ole etukäteen varauduttu tai tahdottu varautua (esimerkiksi tietokantayhteyden äkillinen katkeaminen tai ohjelmistoa suorittavan käyttäjätunnuksen riittämättömän oikeudet kansioon kirjoitusta varten) käsitellään tästä syystä kriittisinä virheinä.

Virheidenkäsittelyn puutteisiin varaudutaan kuitenkin muilla tavoin. Osaprosessien tila tallennetaan aina levylle, jolloin esimerkiksi virhe tietokantaan kirjoittamisessa ei tarkoita sitä, että joudutaan ajamaan koko prosessi uudelleen. Ainoastaan tietokantaan kirjoitus ja mahdolliset kirjoituksen jälkeiset vaiheet on ajettava uudelleen, mikäli virhe on ratkaistavissa ilman ohjelmiston uutta käännöstä tai jos muutos ohjelmistossa koskee ainoastaan kirjoittavaa osaa.

Lisäksi jollain tapaa virheelliset sisään luetut rivit ja mahdollisesti näistä riveihin riippuvaiset rivit tallennetaan alkuperäisessä muodossaan omiin tiedostoihinsa ja ilmoitetaan osaprosessin tulosten kautta käyttäjälle. Näin virheelliselle sisään tulevalle tiedolle voi joko tehdä käsin korjauksia ja yrittää konversiota uudelleen pelkästään tällä tiedolla tai mahdollisesti tallentaa virheellinen tieto kohdejärjestelmän käyttöliittymän kautta.

Käyttöliittymä

Vaikka käyttöliittymä ei sinänsä ole olennainen osa kokonaisuutta, on se kuitenkin osa nykyistä ratkaisua ja vaikuttanut omalta osaltaan toteutetun ratkaisun suunnitteluun. Kuten kuvasta 3 näkyy, on koko sovellus pyritty ratkaisemaan mahdollisimman käyttöliittymäagnostisella tavalla. Prosessi ei ole riippuvainen käyttöliittymästä ja käyttöliittymän riippuvuudet juuri tätä kyseistä toteutusta kohtaan pyrittiin myös minimoimaan (täyteen riippumattomuuteen ei edes pyritty). Tällä tavoin toteutettu runko on uudelleenkäytettävissä niin web-käyttöliittymältä, graafiselta käyttöliittymältä, konsolista, web-palveluna kuin tavallisena windows palveluna.

Käyttöliittymää alettiin toteuttaa vasta, kun prosessi oli valmis testattavaksi. Ensimmäinen versio oli *WinForms* -toteutus (graafinen käyttöliittymä), mutta jo kohtalaisen pian tämäntyyppinen toteutus alkoi aiheuttaa ylimääräistä vaivaa toteutettavan työn kokoon ja käyttötarkoitukseen nähden. Monimutkaisuutta lisäsi esimerkiksi käyttöliittymäelementtien tilojen hallinta ja käyttöliittymän pitäminen reaktiivisena prosessin suorituksen ajan. Tämän jälkeen siirryttiin kokeilemaan konsolikäyttöliittymää, josta kohtalaisen

pienellä vaivalla saatiinkin toteutettua kevyt versio, johon on helppo lisätä toiminnallisuutta (nykyisessä versiossa on esimerkiksi toiminto, joka poistaa tietokannan ja luo uuden tilalle) ja joka on edelleen käytössä.

6.4 Konversion toteuttavan kirjaston toteutus

Sisäänluku

Kaikki sisään luettu tieto viipaloidaan särmikkääseen (jagged) string -tyyppiseen taulukkoon. .NET Frameworkissa kaksiulotteinen taulukko tarkoittaa eri tietotyyppiä kuin yleisesti tietojenkäsittelyssä. Särmikäs taulukko on taulukko, jota yleisesti kutsutaan moniulotteiseksi taulukoksi, jonka jokainen alkio on myös taulukko. Kaksiulotteinen taulukko on taas .NET:ssä taulukko, jonka jokainen alkio muodostuu kahdesta samaa tietotyyppiä olevasta tiedosta. [15; 16.]

Ylimmän tason taulukko vastaa tietueen tyyppiä, seuraavan tason taulukko vastaa yksittäistä tietuetta ja alimman tason taulukon alkiot ovat yhden tietueen konversioon tarvittavat tiedot. Sisään luettavaa tietoa siis puhdistetaan jo lukemisen aikana, lisäksi tarkemmalla viipaloinnilla varmistetaan se, että sisään luetuissa riveissä on oikeasti tietoa niin paljon kuin mitä kuuluukin. Tämä edistää virheidenhallinnan lähtökohtaa, jossa virheet pitäisi huomata mahdollisimman aikaisessa vaiheessa.

Tiedon muunnos

Tiedon muunnos on konversion olennaisin vaihe ja ehdottomasti myös toteutuksen kannalta kiinnostavin osa-alue. On tärkeää, että kaikki olennainen tieto tulee talteen, on sopivaa kohdejärjestelmän kannalta ja että viite-eheyksistä huolehditaan. Osittain tästä syystä muunnos on myös ainut implementaation vaihe, jossa on enemmän kuin kaksi luokkaa käytössä ja jota refaktoroitiin paljon toteutuksen aikana. Koska vaiheen onnistuminen on tärkeää, on myös tärkeää, että vaihe on pilkottu riittävän selkeisiin osiin, jotta mahdolliset virheet on helppo paikallistaa eivätkä korjaukset heijastele muutoksia pitkin osaprosessia.

Muunnoksessa viipaloitu tieto kohdistetaan ja muunnetaan tietuetyyppi ja tietue kerrallaan valmisteltuun tietokantaa vastaavaan tietorakenteeseen. Koska muunnoksen jälkeinen tietorakenne on sama kuin tietokannassa, kaikki tietotyyppien muunnosvirheet tulevat ilmi jo tässä vaiheessa.

Aluksi jokainen osamuunnos ja niiden tarvitsemat metodit olivat samassa luokassa. Luokan alkaessa paisua ja metriikan osoittaessa liiallista monimutkaisuutta jokaisen tietueen muunnos ja siihen liittyvät metodit pilkottiin omiin luokkiinsa, joita käytettiin Tehdas-suunnittelumallin mukaisesti [*Factory* 17, s.107–117]. Osaprosessi ei ole yksinkertaisuuden huipentuma ja asioita joudutaan tekemään pitkälti kohdejärjestelmää palvellen, mutta metriikan mukaan luokassa ei ole mitään mullistavan monimutkaista (jokaisen metodin syklomaattinen kompleksisuus enintään 5).

Muunnetun tiedon tallennus

Koska tietueet oli sovitettu tietokannan rakenteisiin jo aiemmissa vaiheissa, oli itse tiedon tallentaminen helppoa ja suoraviivaista. Tiedon kirjoittamiseen käytettiin .NET Frameworkin valmiita luokkia, joiden avulla pystyttiin tallentamaan tietoa taulu kerrallaan. Osaprosessin tärkeimmäksi tehtäväksi jäikin lopulta lähinnä tallennettavan tiedon tallennusjärjestyksestä huolehtiminen.

Jokaisen taulun tallennus tapahtuu oman transaktionsa sisällä. Tämä ei välttämättä olisi pakollista kuin ensimmäisen taulun tallennuksen kohdalla, sillä kohdejärjestelmän ja sen tietokannan rakenteesta ja toiminnallisuudesta johtuen virhetilanteessa on parempi alustaa tietokanta uudelleen, mikäli vain osa tiedosta tallentuu oikein. Koska transaktioiden käytöstä ei kuitenkaan ole haittaa, niiden käyttöä ei lähdetty erittelemään taulukohtaisesti.

Muunnetun tiedon tarkistukset

Tiedon tarkistusten osaprosessi (validointi) esitellään tarkoituksella tietokantaan kirjoittamisen jälkeen, vaikka kuvassa 2 tarkistus on esitetty tapahtumaan ennen kirjoitusta. Tämä johtuu siitä, että osaprosessi lisättiin ratkaisuun kesken kehityksen, sillä sen si-

sältämät toimenpiteet eivät tuntuneet täysin osuvan niin muunnokseen kuin kirjoitukseenkaan.

Siinä missä muunnoksessa otetaan lähinnä huomioon viite-eheyteen liittyvät tietojen riippuvuudet, tässä osaprosessissa käsitellään tietueiden sykliset riippuvuudet, jotka johtuvat siitä, että tietueen 2 sisältämä tieto koostuu tietueen 3 sisältämän tiedon summista. Lisäksi tarkistuksissa luodaan testausta ja tuotantoon siirron varmistusta varten CSV-muotoinen tarkistuslista, johon kootaan kohdejärjestelmän käytön ja muunnetun tiedon kannalta olennaisia tietoja ja tietojen summia. Näin myös ei-tekniset ja liiketoimintaa tuntevat henkilöt pystyvät tarkistamaan konversion tuloksia helposti esimerkiksi lataamalla tuotettu tiedosto Exceliin.

Tietueiden tietosisältöjen riippuvuudet olisi voitu hoitaa jollain muulla tavalla, etenkin kun ne jäivät vähälle huomiolle alun alkaenkin. Summaamisen olisi voinut hoitaa esimerkiksi konversion jälkeen muutamalla T-SQL -kyselyllä. Tällöin olisi kuitenkin menetetty konversioon liittyvät tarkistus- ja raportointimahdollisuudet ja kasvatettu epäonnistumisen riskiä ylimääräisillä manuaalisilla askeleilla.

Tämän osaprosessin erottaa muista myös laaja *Language Integrated Queryjen* (LINQ) käyttö. LINQ mahdollistaa kyselyiden, järjestelyn tai suodatuksien kirjoittamisen melkein mille tahansa .NET-oliokokoelmalle [18, s.144]. Siinä missä muut ohjelmiston osat ovat onnistuneet helposti ilmankin, tarkistuksissa ja summien laskemisissa LINQ on työkaluna paikallaan. Normaalisti olisi tarvittu useamman rivin pituisia metodeja apumuuttujineen jokaisen erilaisen summan laskemiseen, mutta tällä hetkellä mikään kysely ei ole kolmea riviä pidempi ja todennäköisesti koodi on myös selkeämpää niille jotka ymmärtävät LINQ:n syntaksia. Lisäetuna oli myös tekemisen nopeus, tarkistusten ja summaamisten kirjoittamiseen kului aikaa noin yksi työpäivä.

Prosessiosien analysointi

Osaprosessin onnistumisen analysointi on jaettu kahteen eri osioon, kuten jo aiemmin on tullut ilmi. Ensimmäinen osio ajetaan aina ennen itse osaprosessia, jolloin voidaan tarkistaa, onko olemassa edellytyksiä osaprosessin ajamiselle. Toinen osio tarkastelee prosessissa suoritettuja toimenpiteitä ja mahdollisesti myös tarkistaa toimenpiteiden jälkeisiä tuloksia. Jokainen osaprosessi ei välttämättä analysoi mitään, vaikka jokainen

osaprosessi toteuttaakin analyysirungon. Näin mahdollisille tarkistusten myöhäisille ideoille on valmiina paikka, johon ne on yksinkertaista toteuttaa.

Ensimmäisen osion tulosta käytetään osaprosessin suorituksen keskeyttämiseen, mikäli jokin ennakkoehto prosessin ajolle ei täyty. Esimerkiksi tietoa luettaessa tarkistetaan, että asetuksien kautta annettu sisäänlukukansio sisältää tiedostoja. Mahdollisia muita tarkistuksia voisi olla kirjoitusvaiheessa tietokantayhteyden toiminnan tarkistaminen tai kaikkien osaprosessin käyttämien asetustietojen tarkistus.

Mikäli esianalyysissa annetaan prosessoinnin jatkua, toisen analyysiosan tulokset ylikirjoittavat ensimmäisen osan tulokset. Analyysitulokset kirjoitetaan joka ajokerta lokiin, jotta voidaan todeta tietynasteinen onnistuminen tai epäonnistuminen tutkimatta itse tietokannan tietosisältöä. Toinen osio tarkastelee prosessissa suoritettuja toimenpiteitä ja mahdollisesti myös tarkistaa niiden toimenpiteiden jälkeisiä tuloksia. Esimerkkinä muunnoksen jälkeinen analyysi tarkastelee levyltä luettujen rivien määrää ja vertailee saatua lukua tietuetta vastaavan DataTable-olion sisältämää rivimäärää.

6.5 Testaus

Tuotetun ohjelmakoodin yksikkötestaus jäi erittäin vähäiseksi. Tämä johtui ainoastaan tottumattomuudesta yksikkötestauksen osalta, sillä toteutuksen ja toteutuksessa käytettävän ohjelmistotekniikan metodologian suhteen oli täysi valinnan vapaus. Ainut kohta koodia, jossa todella hyödynnettiin yksikkötestausta, oli erään kohdejärjestelmästä C#:lle käännetyn muunnosalgoritmin toiminnan testaus.

Muutoin sovellusta testattiin niin sisäisesti kuin asiakkaallakin käyttäen sekä konversion tuottamia tarkistuslistoja että kohdejärjestelmään tallennettua tietoa. Eniten virheitä löytyi käyttämällä kohdejärjestelmää konvertoiduilla tiedoilla ja useimmiten virheet liittyivät tiedon oikeantyyppiseen kohdentamiseen, joka ilmeni vasta kun kohdejärjestelmä ei toiminut oletetulla tavalla.

7 Työn ja metriikan tulosten analysointi

7.1 Mitatut arvot

Mitatut arvot edustavat kahta eri ohjelmiston kehityksen osa-aluetta. Visual Studion mittaamat arvot antavat suuntaa matalan tason toteutuksen ongelmissa. Näillä arvoilla suurin saatu hyöty saadaan lyhyellä aikavälillä, sillä niiden kautta saadaan nopeasti selville liian monimutkaiset sekä pitkät metodit ja rutiinit. Toki lyhyet ja yksinkertaiset metodit auttavat myös pidemmällä aikavälillä, sillä silloin metodeja on todennäköisesti helpompi liikutella luokkien välillä, mikäli näin päästään parempaan vastuiden yhteenkuuluvuuteen. Tämä on kuitenkin pikemminkin toissijainen hyöty.

Paketti-tason arvot taas kuvaavat korkean tason suunnittelun ominaisuuksia, kuten abstraktiutta (*Abstractness [A]*), riippuvuussuhteita (*Afferent Coupling [Ca], Efferent Coupling [Ce]*), epävakautta (*Instability [I]*) sekä etäisyyttä pääsarjasta (*Distance [D]*) [12, s.427–434]. Näiden ominaisuuksien avulla voidaan päätellä, kuinka helppoa tai haastavaa tietyn paketin ylläpitäminen on, ja löytää refaktoroinnille sopivia kohteita tuntematta ohjelmakoodia ennestään. Kuten jo aiemmin mainittu, paketti-tason arvot mitattiin vasta työn valmistuttua, joten ne toimivat ainoastaan indikaattorina saavutetulle laadulle ja ylläpidettävyyden mahdollisuuksille.

Mittaustuloksissa esiintyvät pakettien nimet tarkoittavat tekstissä aiemmin esiintyneitä termejä seuraavalla tavalla. Core tarkoittaa kirjastopakettia, jossa on määritelty abstraktit luokat ja rajapinnat toteutukselle. Framework on työkalua varten kirjoitettu pienimuotoinen sovelluskehikko. Console on käyttöliittymän sisältävä suoritettava ohjelma, ja Implementaatio on kirjastopaketti, jossa on toteutettu konversioon liittyvät osaprosessit sekä niiden käyttämät luokat, jotka eivät sopineet yleiseen sovelluskehikkoon.

7.2 Mittausten tuloksia

Assemblies	# lines of code	# IL instruction	# Types	# Abstract Types	# lines of comment	% Comment	% Coverage	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
[redacted].Konversio.Core v0.1.4866.21746	6	48	4	4	16	72	-	12	6	1.5	0.33	1	0.24
[redacted].Konversio.Framework v0.1.4866.21748	272	1993	6	0	38	12	-	12	77	1	0.87	0	0.1
[redacted].Konversio.Console v1.0.0.0	103	797	7	1	27	20	-	0	18	2.57	1	0.14	0.1
[redacted].Konversio.[redacted] v0.2.4866.21751 Implementaatio	842	6407	23	1	238	22	-	0	75	1.39	1	0.04	0.03

Kuva 6. Paketti-tason metriikan tulokset toteutetusta ohjelmistosta.

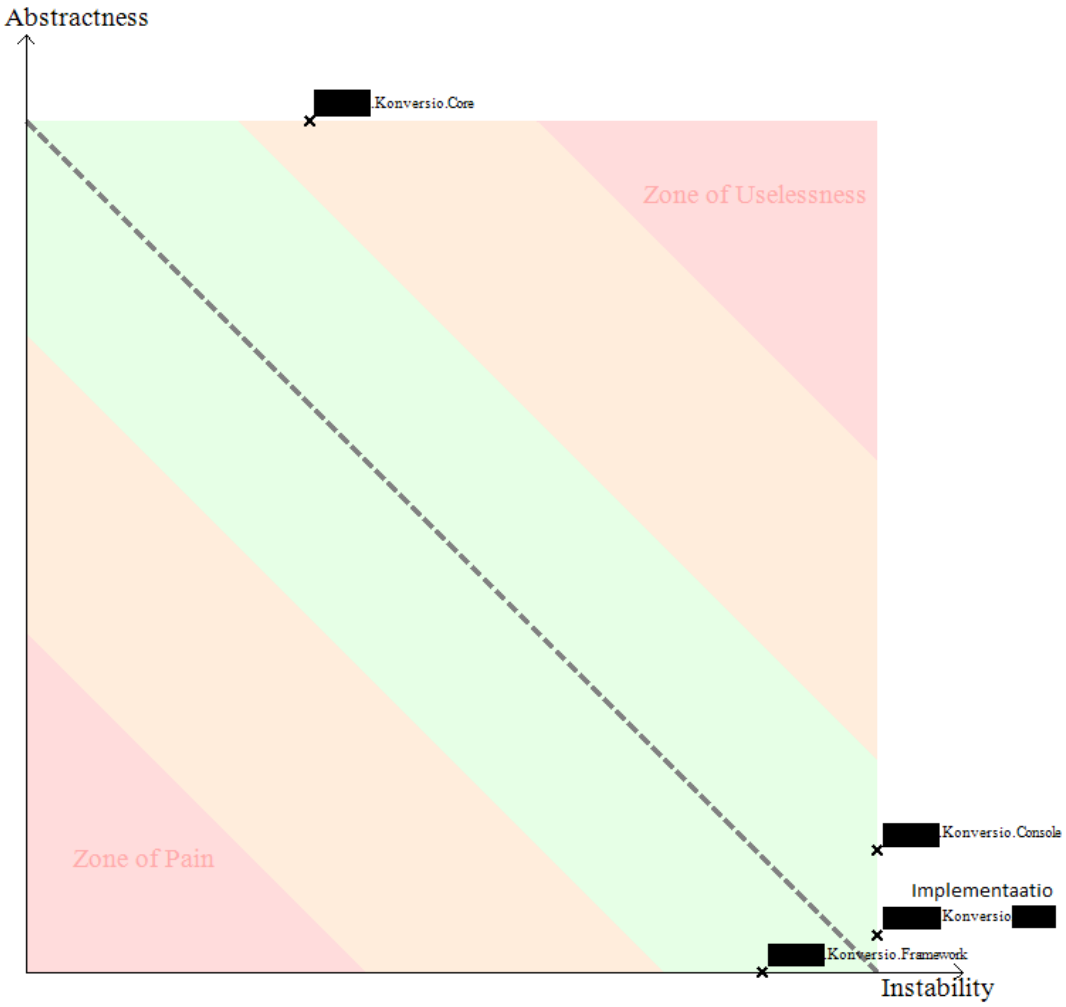
Kuvassa 6 listataan kunkin paketin pakettitason mittareiden tulokset NDepend-työkalua käyttäen. Pakettiin kohdistuva riippuvuus (*Afferent coupling*) merkitsee paketista riippuvien luokkien määrää. Paketin riippuvuus (*Efferent coupling*) taas merkitsee luokkien määrää, joista paketti on riippuvainen (NDepend laskee tähän lukuun mukaan myös .NET Framework-luokkiin olevat riippuvuussuhteet). Epävakaus (*Instability*) johdetaan edellä mainituista kaavalla I = Ce/(Ca+ Ce). Tällöin täysin vakaa paketti ei ole riippuvainen mistään, mutta muut paketit ovat siitä riippuvaisia. Täysin epävakaa paketti taas riippuu muista paketeista, eikä mikään paketti ole siitä riippuvainen. Abstraktisuus (Abstractness) lasketaan paketin sisällä olevien abstraktien luokkien suhteesta konkreettisiin luokkiin. Etäisyys (*Distance*) pääsarjasta (*Main sequence*) johdetaan abstraktiudesta ja epävakaudesta kaavalla D = |A + I - 1| / √2. Etäisyys on kaikista mainituista paras mittari paketin hyödyllisyyden ja ylläpidettävyyden kannalta. [12, s.427–434.]

	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Implementaatio	70	235	2	60	941
Framework	70	133	1	60	312
Core	98	12	1	3	11
Console	75	60	2	16	156

Kuva 7. Visual Studio 2012:lla mitatut metriikan tulokset.

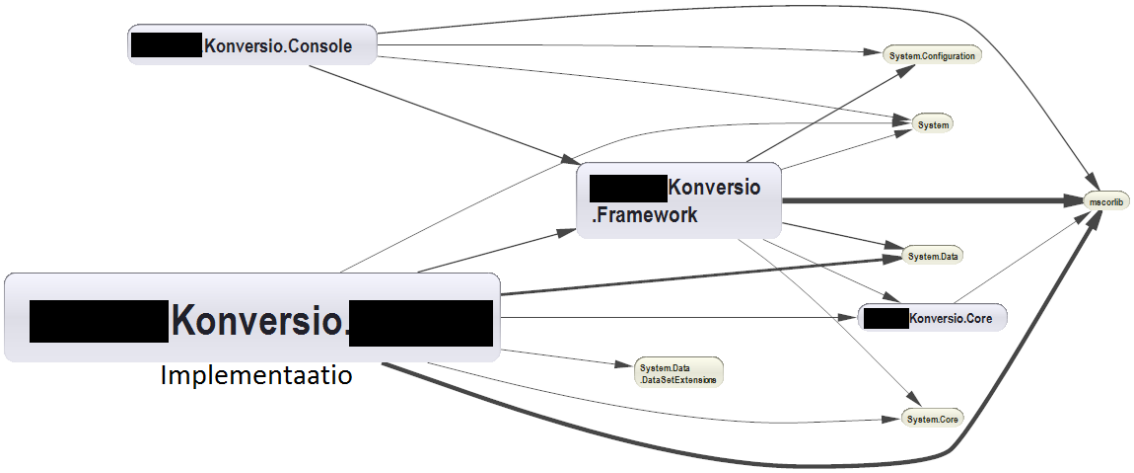
Kuvassa 7 esitellään Visual Studio 2012:n kautta mitatut arvot pakettitasolla laskettuna. Ylläpidettävyysindeksi on painotettu keskiarvo metodi-tasolla lasketuista arvoista. Syklomaattinen kompleksisuus on jokaisen paketissa olevan metodin arvon summa. Sukupuun syvyys on suurin paketissa esiintyvän sukupuun pituus periytyvinä olioina laskettuna. Luokkien riippuvuus kuvaa paketin käyttämien olioiden määrän summaa. Tätä mittaria ei ole käytetty työssä, sillä se ei välttämättä ole riittävän kuvaava mittari laadun kannalta kuten myöhemmin todistetaan. Rivien määrä on paketin sisällä olevien rivien

summan arvio. Mittari on tällä työkalulla arvio siksi, että se lasketaan käännetystä tavukoodista, eikä oikeasti kirjoitetuista riveistä. [20; 21]



Kuva 8. Toteutetut paketit pääsarjaan sijoitettuna.

Kuvassa 8 on jokainen paketti sijoitettu pääsarjalle, joka on lineaarinen suora täysin abstraktista ja vakaasta paketista täysin epävakaaseen ja konkreettiseen pakettiin. Mikäli paketti on kaukana pääsarjasta, on se joko turha (paketissa ei ole konkreettisia luokkia, eikä mikään paketti riipu siitä) tai vaikea ylläpitää (paketti sisältää ainoastaan konkreettisia luokkia ja muut paketit riippuvat siitä). Mitä kauemmaksi pääsarjasta paketti sijoittuu etäisyytensä puolesta, sitä varmemmin paketti kannattaisi ottaa tarkempaan tarkasteluun mahdollista refaktorointia varten. [12, s.432–434.]



Kuva 9. Ratkaisun pakettien väliset riippuvuudet.

Kuva 9 esittää pakettien välisten riippuvuuksien suunnat. Kuvassa on tuotettujen pakettien lisäksi ne .NET Frameworkin ohjelmistokirjastot, joista toteutetut paketit ovat riippuvaisia. Nuolen paksuus kertoo riippuvuuksien määrästä eri luokista ja paketin koko kuvaa sen rivien määrää suhteessa muihin paketteihin (pois lukien kolmannen osapuolen paketit).

7.3 Mittaustulosten analysointi

Metriikan antamat tulokset ovat positiivisia ja vahvistavat tekemisen aikaista käsitystä ohjelmiston ominaisuuksista. Kaikkien pakettien paketti-tason arvot ovat hyvänä pidettyjen raja-arvojen sisällä, konkreettisia luokkia sisältävät paketit ovat enemmän riippuvaisia muista paketeista ja abstrakti kirjasto on riippumaton, mutta siitä ollaan riippuvaisia. Core-kirjasto ei osu yhtä lähelle pääsarjaa kuin muut paketit, mutta se johtunee siitä, että NDepend laskee myös paketin sisäiset riippuvuudet ja riippuvuudet .NET Frameworkin paketteihin, mikä nostaa epävakauden arvoa nollasta.

Myös Visual Studiolla mitatut arvot ovat kohtalaisen hyviä. Esimerkiksi implementaatiossa on NDependillä laskettuna 116 metodia ja tuo arvo yhdistettynä Visual Studion mittauksiin antaa keskiarvoksi 7.2 riviä per metodi ja syklomaattinen kompleksisuus kaksi per metodi. Tottakai variaatiotakin on molempiin suuntiin, mutta arvoihin voi olla tyytyväinen, kun ottaa huomioon että implementaatiossa ei keskitytty laatuun, sillä se on kokonaisuutena kertakäyttöinen. Kehikolle samat arvot olivat 67 metodia, 4.7 riviä per metodi ja syklomaattinen kompleksisuus kaksi per metodi.

Pakettien sijoittuminen niin hyvin pääsarjaan nähden oli hienoinen yllätys. Pyrkimällä noudattamaan SOLID-periaatteita niin matalalla kuin korkeallakin abstraktion tasolla saatiin tuotettua ohjelmisto, joka myös käytettyjen mittarien mukaan on tietyiltä osin uudelleenkäytettävissä.

Metriikan käytön kanssa tulee olla myös kriittinen, ja asioiden todellisen laidan näkee ainoastaan lukemalla koodia. Tiedon muuntamisen osaprosessin sisällä on muutama ohjelmakoodin kohta, joissa pelkkä metriikan tuijottaminen saattaisi johtaa vääriin johtopäätöksiin. Tiedonsiirtoa varten luotavan DataSet-olion alustus tehdään yhden metodin sisällä ja metodin ylläpidettävyysindeksi on 22/100. Tälle ei sinänsä voi mitään, sillä jossain kyseisen olion alustus tulee tehdä, ja koska se tehdään käsin, näkyy se metriikassa koodin yleistä laatua laskevana tekijänä, ellei välttämättä haluta päästä johonkin tiettyyn metriikka-arvoon esimerkiksi metodia pienempiin osiin pilkkomalla. Toinen virheelliseltä näyttävä metodi on erään tietueen tiedon muunnosta varten kirjoitettu apumetodi, jonka syklomaattinen kompleksisuus on 22. Yleisesti ottaen liian monimutkaisen metodin rajana pidetään 15:ä. Kyseessä on kuitenkin kokonaisluku-kokonaisluku-muunnos, koska samaa asiaa esittävä numeerinen arvo tarkoittaa lähdejärjestelmässä eri arvoa kuin kohdejärjestelmässä. Metodi on siis 22-kohtainen switch-lause, joka ei varmastikaan ole kenellekään ohjelmoinnin perusteita suorittaneelle millään tavalla monimutkainen ymmärrettävä.

7.4 Suunnittelman analysointi

Nykyinen suunnitelma kestää vaatimusten muutosta hyvin. Oiva esimerkki siitä oli yhden prosessiosan lisääminen kesken kehityksen, mikä ei tuottanut minkäänlaisia vaikeuksia. Ei kuitenkaan voi suunnitella liian pitkälle etukäteen ja varmasti toisenlaisen toteutuksen tekeminen kirjastoa käyttäen antaisi lisää näkemystä niihin kohtiin, jotka ovat yhteisiä ja nostaisi osia tästä implementaatiosta kirjastotasolle yleishyödyllisyytensä vuoksi. Työ suoritettiin myös itsenäisesti, ja on todennäköistä, että useampi näkökulma olisi muokannut ohjelmistoa monistakin kohdista. Olennaista on se, että tehty työ täyttää sille asetetut vaatimukset.

Tällä hetkellä Framework-moduulilla on kaksi vastuuta, mutta koska tämä ei johda suoraan sykliseen riippuvuuteen [kuvat 5 ja 9], olisi ProcessManager-luokan erottaminen

omaksi kirjastokseen lähinnä akateeminen harjoitus, joka ei toisi tällä hetkellä sovelluksen toimintaan mitään käytännön hyötyä.

7.5 Osaprosessien analysoinnin tarkastelu

Sovelluksessa tapahtuvan analysoinnin suunnittelu ja toteutus ovat jälkikäteen olleet eniten askarruttavia ja epäilystä aiheuttaneita osioita. Yhtä hyvin analysoinnin olisi voinut jättää täysin toteutuksen harteille joko toteutettavaksi omana metodinaan, erillisenä luokkanaan tai omana erillisenä osaprosessinaan. Toisaalta myös jokaisessa edellä luetellussa vaihtoehdossakin on omat ongelmansa. Mikäli jokainen analyysivaihe olisi oma osaprosessinsa, saattaisi kokonaisprosessin punainen lanka kadota toteutusta tehdessä (mikä osaprosessi vastaa mistäkin). Jos taas analyysi toteutettaisiin osaprosessin sisällä metodina tai omana luokkanaan, laajentaisi se itse osaprosessin vastuuta muun toiminnallisuuden lisäksi omien ja aiemmin suoritettujen prosessiosien tulosten tarkistukseen ja analysointiin.

Tässä kontekstissa vahvin peruste ottaa analysointi mukaan abstraktiokirjastoon ja sitoa se läheisesti osaprosesseihin on analysoinnin vaatimuksen toistuminen osaprosessista osaprosessiin. Kirjastoa suunniteltaessa tärkeä lähtökohta oli nimenomaan etsiä eri osaprosesseja yhdistäviä tekijöitä. Analyysi jaettiin kahteen osaan siksi, että jokainen osaprosessi pysyisi mahdollisimman pitkälle puhtaana muiden osaprosessien tiedoista. Mikäli esimerkiksi sisäänluvun loppuanalyysi tarkistaisi asioita, jotka ovat muunnoksen aloitukselle tärkeitä, muunnoksen muuttuessa muutos voisi heijastua sisään luvun analyysiin. Kun analyysi on jaettu kahteen vaiheeseen, osaprosessin muuttuessa ainoastaan kyseisen osaprosessin analyysi muuttuu mukana.

7.6 Refaktoroinnista

Joka paikkaa, edes tehtyä kehikkoa ei ole refaktoroitu loppuun asti. Tämä johtuu siitä, että tehtävän työn luonne on osittain kertakäyttöinen ja työtä varten varattu aika rajallinen.

```
private void LoadProcessPart(string processPartName, int index)
{
 foreach (Type t in implementation.GetTypes())
 {
```

```
  if (t.IsClass)
  {
   // Initialize Transferobject only once
   if (transfer == null)
    if (t.BaseType.Equals(typeof(TransferObject)))
     transfer = (TransferObject)Activator.CreateInstance(t);

   // Initialize process parts and corresponding analyzers
   if (t.Name.Contains(processPartName))
   {
    if (t.BaseType.Equals(typeof(AbstractProcess)))
      processParts[index] =
            AbstractProcess)Activator.CreateInstance(t)
    if (t.GetInterface(typeof(IAnalyzer).Name) != null)
     analyzeParts[index] = (IAnalyzer)Activator.CreateInstance(t);
   }
  }
 }
}
```

Esimerkkikoodi 1. ProcessManager-luokan yksittäisen osaprosessin latauksen koodi.

Esimerkkikoodissa 1. oleva ProcessManager-luokan LoadProcessPart-metodi voitaisiin pienellä vaivalla jakaa neljäksi omaksi metodikseen. Näin SRP:tä noudatettaisiin tarkemmin.

```
private void LoadProcessPart(string processPartName, int index){
 foreach (Type t in implementation.GetTypes()){
  if (t.IsClass){
   // Initialize process parts and corresponding analyzers
   if (t.Name.Contains(processPartName)){
    AddProcessPart(t);
    AddProcessAnalyzer(t);
   }
  }
 }
}

// Tätä metodia kutsuttaisiin Initializen sisällä vain kerran
private void TransferObject(){
 foreach (Type t in implementation.GetTypes()){
  if (t.IsClass){
    if (t.BaseType.Equals(typeof(TransferObject)))
     transfer = (TransferObject)Activator.CreateInstance(t);
  }
 }
}

private void AddProcessPart(Type t){
 if (t.BaseType.Equals(typeof(AbstractProcess)))
  processParts[index] = AbstractProcess)Activator.CreateInstance(t);
}

private void AddProcessAnalyzer(Type t){
 if (t.GetInterface(typeof(IAnalyzer).Name) != null)
  analyzeParts[index] = (IAnalyzer)Activator.CreateInstance(t);
}
```

Esimerkkikoodi 2. ProcessManager-luokan yhden osaprosessin latauksen koodi refaktoroituna.

Syy miksei koodia ole refaktoroitu esimerkkikoodin 2 tavalla on se, että prosessien lataus on toiminut oletetusti ensimmäisestä kirjoituksesta alkaen eikä tästä johtuen ole ollut tarvetta käydä ohjelmakoodia näiltä osin tarkemmin läpi. Kaikki koodi käytiin läpi uudelleen erityisesti mahdollisia refaktorointia vaativia kohtia etsien opinnäytetyötä varten niin lähellä tuotantoon vientiä, että ei ollut syytä muuttaa ohjelmistosta mitään toimivaa osuutta [12, s.66]. Aina löytyy jotain kehitettävää ja parannettavaa, mutta on myös osattava jossain vaiheessa toistaiseksi tyytyä toimivaan ratkaisuun.

8 Päätelmät

Suunniteltu runko ja sovelluskehikko mahdollistavat monia asioita. Samalla rungolla pystyttäisiin toteuttamaan tietuepohjaiseen strategiaan pohjautuvaa konversio, sillä runko itsessään ei näe kuin prosessin. Ainut nähtävissä oleva lisäys tai muutos olisi eräkoon lisääminen ohjelmiston asetuksiin.

Vaikka työlle ei annettu tarkkoja tai tiukkoja ajallisia raameja valmistumisen suhteen, työ ei missään vaiheessa pidätellyt ympärillä pyörivän projektin etenemistä. Sekä tekniset että konversioon liittyvät suunnitelmat kantoivat loppuun saakka suurimmilta osin, eikä korjattavien asioiden listalle päätynyt mitään kriittistä osaa. Toteutettu implementaatio suoriutuu kaiken tiedon (noin puoli gigatavua) konvertoinnista sekä muista tavoitteista (tarkistuslistojen generointi) annetussa aikarajassa (mielellään alle kaksi tuntia, tulevassa käyttöympäristössä suoritusaika jäi puoleen tuntiin), joka mahdollistaa järjestelmän päivityksen normaalin huoltokatkon aikana.

Vaikka toteutetulle ohjelmiston jatkokehitykselle ei ole olemassa suunnitelmia, koen että nähty vaiva suunnittelussa kannatti. Mahdollista seuraavaa samantapaista työtä varten on olemassa hyvin suunniteltu ja koeteltu runko. Lisäksi pitkä ja monitahoinen suunnittelu ja suunnittelun dokumentointi on kehittänyt minua ohjelmoijana monilta osin.

Lähteet

1 Data Migration. 2011. Verkkodokumentti. < http://www.dataintegration.info/data-migration>. Luettu 12.10.2012.

2 Data Migration. 2013. Verkkodokumentti. <http://en.wikipedia.org/wiki/Data_migration>. Luettu 14.3.2013.

3 An Introduction to ETL. 2012. Verkkodokumentti. <http://www.etlsolutions.com/an-introduction-to-etl/>. 5.12.2012. Luettu 15.1.2013.

4 Passioned Group. List of ETL Tools. 2013. Verkkodokumentti <http://www.etltool.com/list-of-etl-tools/>. Luettu 25.2.2013.

5 Mundy, Joy. Should You Use an ETL Tool? 2006. Verkkodokumentti. <http://www.informationweek.com/software/business-intelligence/kimball-university-should-you-use-an-etl/207002081>. 6.4.2008. Luettu 12.10.2012.

6 Becker, Bob. Six Key Decisions for ETL Architechtures. 2009. Verkkodokumentti. <http://www.kimballgroup.com/2009/10/09/six-key-decisions-for-etl-architectures/>. 9.10.2009. Luettu 12.10.2012.

7 Extract, Transform, Load. 2013. Verkkodokumentti. <http://en.wikipedia.org/wiki/Extract,_transform,_load>. Luettu 12.10.2013.

8 Javlin, Inc. ETL (Extract-Transform-Load) 2011. Verkkodokumentti. <http://www.dataintegration.info/etl>. Luettu 7.1.2013.

9 Sommerville, Ian. 2010. Software Engineering. Pearson Education; 9th International Edition.

10 EU:n tietosuojadirektiivi 1995. Verkkodokumentti. < http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:fi:NOT>. Luettu 30.3.2013.

11 Kimball, Ralph. The 38 Subsystems of ETL. 2004. Verkkodokumentti. <http://www.informationweek.com/the-38-subsystems-of-etl/55300422>. 4.12.2004. Luettu 12.10.2012.

12 Martin, Robert C. & Martin, Micah. 2006. Agile Principles Patterns and Practices in C#. Westford, Massachusetts: Prentice Hall.

13 Fowler, Martin. 1999. Refactoring: Improving the Design of Existing Code. Westford, Massachusetts: Addison-Wesley Professional.

14 Hickey, Rich. Hammock-driven Design. 2010. Verkkovideo. <http://www.youtube.com/watch?v=f84n5oFoZBc >. Katsottu: 15.11.2012.

15 Taulukko (tietorakenne). 2013. Verkkodokumentti. <http://fi.wikipedia.org/wiki/Taulukko_(tietorakenne)>. Luettu 31.3.2013.

16 Microsoft Corporation. Jagged Arrays (C# Programming Guide). 2013. Verkkodokumentti. <http://msdn.microsoft.com/en-us/library/2s05feca.aspx>. Luettu 31.3.2013.

17 Gamma, Helm, Johnson & Vlissides. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Westford, Massachusetts: Addison-Wesley.

18 Johnson, Glenn. 2011. MCTS Self-Paced Training Kit (Exam 70-516): Accessing Data with Microsoft® .NET Framework. Microsoft Press.

19 NDepend. 2013. Visual Studio-lisäosan kotisivut. Verkkodokumentti. <http://www.ndepend.com/Default.aspx>. Luettu 25.4.2013.

20 Microsoft Corporation. Code Metrics Values. 2013. Verkkodokumentti. <http://msdn.microsoft.com/en-us/library/bb385914.aspx>. Luettu 25.4.2013.

21 Steve St.Jean, MVP. Visual Studio 2010–New and little-know features–Part 2–Code Metrics. 2012. Verkkodokumentti. <http://sstjean.blogspot.fi/2012/04/visual-studio-2010new-and-little-know_09.html>. Luettu 25.4.2013.

ProcessManager.cs

```
public class ProcessManager
{
 //prosessin tarvitsemien luokkien abstraktiot
 private IAnalyzer[] analyzeParts;
 private AbstractProcess[] processParts;
 private TransferObject transfer;

 //prosessin toteuttava moduuli
 private Assembly implementation;

 //Lista osaprosessien nimistä
 private List<string> processPartNames;


 public void Initialize(string assemblyName, string processOrder)
 {
  LoadImplementationAssembly(assemblyName);

  processPartNames = processOrder.Split(new char[] { ',' }.ToList();

  processParts = new AbstractProcess[processPartNames.Count];
  analyzeParts = new IAnalyzer[processPartNames.Count];

  for (int i = 0; i < processPartNames.Count; i++)
  {
   LoadProcessPart(processPartNames[i], i);
  }
 }

 private void LoadImplementationAssembly(string assemblyName)
 {
  implementation = Assembly.LoadFrom(assemblyName);
 }

 private void LoadProcessPart(string processPartName, int index)
 {
  foreach (Type t in implementation.GetTypes())
  {
   if (t.IsClass)
   {
    // Initialize Transferobject only once
    if (transfer == null)
     if (t.BaseType.Equals(typeof(TransferObject)))
      transfer = (TransferObject)Activator.CreateInstance(t);

    // Initialize process parts and corresponding analyzers
    if (t.Name.Contains(processPartName))
    {
     if (t.BaseType.Equals(typeof(AbstractProcess)))
       processParts[index] =
             (AbstractProcess)Activator.CreateInstance(t)
     if (t.GetInterface(typeof(IAnalyzer).Name) != null)
      analyzeParts[index] = (IAnalyzer)Activator.CreateInstance(t);
    }
   }
  }
 }
}
```